BIG DATA

BIG DATA

爽！可以自己亲手搭建并有效管理大数据平台了！

大数据

从基础理论到最佳实践

主　编　祁　伟

副主编　刘　冰　常志军　赵廷涛　高俊秀



清華大学出版社

大数据：从基础理论到最佳实践

祁　伟　主　编
刘　冰　常志军　赵廷涛　高俊秀　副主编

清華大学出版社
北　京

内 容 简 介

本书侧重于大数据的实践性技术，系统地介绍了主流大数据平台及工具的安装部署、管理维护和应用开发。平台和工具的选择均为当前业界主流的开源产品，因此，对于读者来说，有很强的可操作性。

本书涉及的开源技术包括：HDFS、MapReduce、YARN、Zookeeper、HBase、Hive、Sqoop、Storm、Kafka、Flume 等。除介绍一般性的背景知识、安装部署、管理维护和应用开发技术外，还特别注重案例实践，重要的技术点以实际工作场景或案例为依托，使读者能快速入门，参考案例动手实践，通过具体深入的实践，体会大数据的技术本质特征，领略大数据技术带来的创新理念，更好地理解和把握信息技术的发展趋势。

本书主要内容包括以下几大部分。

大数据存储篇：以 HDFS 为基础，介绍分布式文件系统的原理、安装、fs 命令的使用、编程，介绍如何用 HDFS 实现，并通过 HTTP 调用。

大数据计算篇：以 MapReduce、YARN 为基础，介绍分布式计算的原理、部署，以及编程案例。

非关系型数据库篇：以 HBase 为基础，重点介绍非关系型数据库的优势、原理、部署，以及命令行使用，编程案例，与 Sqoop 配合使用等。

大数据仓库篇：以 Hive、数据仓库等为基础，重点介绍数据的抽取、原理、部署、分析与编程。

大数据实时计算篇：以 Storm、Kafka 为基础，介绍实时计算的架构、组成、使用与开发。

本书非常适合从事大数据技术开发与使用的初学者，以及从事大数据技术研发的企事业单位工程师学习和参考，也适合高校计算机相关专业的专科生、本科生和研究生学习使用。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

大数据：从基础理论到最佳实践/祁伟主编. —北京：清华大学出版社，2017
ISBN 978-7-302-45743-5

Ⅰ. ①大…　Ⅱ. ①祁…　Ⅲ. ①数据处理　Ⅳ. ①TP274

中国版本图书馆 CIP 数据核字(2016)第 290300 号

责任编辑：杨作梅
装帧设计：杨玉兰
责任校对：张　瑜
责任印制：
出版发行：清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　张：**21.5　　**字　数：**499 千字
版　　次：2017 年 1 月第 1 版　　.　　**印　次：**2017 年 1 月第 1 次印刷
印　　数：1～3000
定　　价：59.80 元

产品编号：069305-01

前　言

技术革命的浪潮推动着人类文明的发展。

第一次浪潮造就了农业革命，它在数千年前出现并持续了数千年；第二次浪潮造就了工业革命，它在数百年前出现并持续了数百年；我们今天正在经历着信息技术第三次浪潮，发端于数十年前，目前也只是处在初级阶段。

农业技术革命释放了“物之力”；工业技术革命释放了“能之力”，而今天的信息技术革命释放的是“智之力”。

距今 400 年前，培根在《伟大的复兴》中预言：知识就是力量。今天，人类终于迎来“知识经济时代”，它是人类社会经济增长方式与经济发展的全新模式。

人类认识物质世界、人类社会和精神世界的最高境界是智慧，而要达智慧的境界，必然要跨越数据、信息、知识三个层级。

数据作为基础，是信息之母、知识之初、智慧之源。正是今天的大数据技术，引燃了人们实现智慧城市、智慧医疗、智慧教育等有关人工智慧的激情。人们真切地认识到，对于人工智能，只要让数据发生质变，即使是简单的数据，也比复杂的算法更有效。

今天，移动互联网的发展，使我们在获取数据上有了质的飞跃，人类的各种社会活动都与互联网这个虚拟世界相联系，使全样本、全过程地有效测量和记录成为可能，构建了生成大数据生态的土壤，同时，人们还在期待和憧憬物联网带来更大的冲击。

另一方面，云计算发展到今天，不论从技术到产业都开始进入成熟期，这也是大数据发展的基石和推进器。

在今天这个时代中，运用大数据洞见事物蕴藏的“智慧”成为人们的渴望。大数据更新了人们对数据的认识。在技术层面，小数据时代的很多数据处理方法和工具已不再有效，需要一系列新的方法和工具。所幸，有大量平民化的开源软件可用，它们不需要特殊的硬件系统，也更适用于云计算环境。

本书正是一本介绍主流的大数据开源软件平台和工具的技术专著，侧重于大数据的实践性技术，帮助读者快速入门，通过具体深入的实践，体会大数据的技术本质特征，领略大数据技术带来的创新理念，更好地理解和把握信息技术的发展趋势。

本书定位

(1) 信息发展已步入大数据时代，当前对于大数据还缺乏面向公众的技术实践手册。

(2) 本书的创作团队有丰富的大数据规划、开发、运营等经验，多位作者成功地架构了教育部、科技部、互联网等大数据架构与分析项目。

(3) 本书的参与者均是部委信息一线工程师、著名外企架构师、国内企业资深高级工程师，所做的理论分析易于学习，实践具有可操作性。

(4) 本书重点介绍大数据的基础理论、关键技术，以及编程实践。利用本书，就可以完全搭建并能有效地管理好大数据平台。

本书特色

(1) 理念先进：均是国内外最新的大数据理念；方便读者全面了解国内外大数据研究与发展的情况。

(2) 技术领先：参与者均是国内 IT 人士；采用的平台均是业界主流开源平台，涉及大数据常用的 HDFS、MapReduce、YARN、Zookeeper、HBase、Hive、Sqoop、Storm、Kafka 等技术的介绍与编程使用。

(3) 案例丰富：提供翔实的实例与解决方法，供项目中参考。

(4) 资源齐备：本书涉及的配套下载资源可以从清华大学出版社的网站中下载。

全书关键字

大数据、分布式计算、数据仓库、数据分析、HDFS、MapReduce、YARN、Zookeeper、HBase、Hive、Sqoop、Storm、Kafka。

由于编者的水平有限，书中难免有疏漏和错误，希望业内专家和广大读者指正。

编　者

目　　录

大数据存储篇

大数据计算篇

非关系型数据库篇

大数据仓库篇

大数据实时计算篇

大数据存储篇

第 1 章

概述

学习目标

本章通过对大数据技术的简要概述，使读者了解大数据的基本概念，以及在分布式环境下，大数据存储和处理的基本原则，掌握大数据主流技术分布，了解大数据可能的职业发展方向，为后续的学习提供指引。

1.1　什么是大数据

有观点认为，人类过去经历了三次工业化技术革命，从蒸汽机时代，到电力时代，再到早期的计算机时代，每一次革命都释放了巨大的生产力，开创了工业的转型和经济的增长时期。人们都说，现在人类正在经历第四次技术革命，数据就是新的源动力。

的确，我们已经看到了海量数据的爆炸式增长景观，特别是来自云端的数据。云端提供了前所未有的计算能力和数据存储能力。这表明，我们已身处“大数据”时代。

但是，关于大数据的确切定义，目前尚未获得统一、公认的说法。

IBM 用 3V(Volume、Variety、Velocity)来描述大数据所拥有的特点。

大容量(Volume)，是指数据体量巨大。

多形式(Variety)，是从数据的类型角度来看的，数据的存在形式从过去的以结构化数据为主转换为形式多种多样，既包含传统的结构化数据，也包含可便于搜索的半结构化数据，如文本数据，还包含更多的非结构化数据，如图片、音频和视频数据。

高速率(Velocity)则是从数据产生效率的实时性角度来衡量的，数据以非常高的速率产生，比如大量传感器生成的实时数据。

之后，IBM 又在 3V 的基础上，增加了 Value 这个维度，即价值密度低的数据称为大数据，意指大数据伴随着从低价值的原始数据中进行深度挖掘和计算，从海量且形式各异的数据源中抽取出富含价值的信息。

由此可以看出，从具备 4V 特性的大量数据中挖掘高价值知识，是各界对于大数据的一个共识。

由于数据量的爆炸式增长，传统的数据管理模式及工具已不能高效地存储和处理如此规模的数据。新时代呼唤新思维、新技术。从维克多 • 迈尔 • 舍恩伯格所著的《大数据时代》中，可以看到大数据时代的思维变革。

(1)　不是随机样本，而是全体数据。

统计学家们证明：采样分析的精确性随着采样随机性的增加而大幅提高，但与样本数量的增加关系不大。随机采样取得了巨大的成功，成为现代社会、现代测量领域的主心骨。但这只是一条捷径，是在不可收集和分析全部数据的情况下的选择，它本身存在许多固有的缺陷。大数据是指不用随机分析法这样的捷径，而采用所有数据的方法。

(2)　不是精确性，而是混杂性。

数据多比少好，更多数据比算法系统更智能还要重要。社会从“大数据”中所能得到的益处，并非来自运行更快的芯片或更好的算法，而是来自更多的数据。大数据的简单算法比小数据的复杂算法更有效。大数据不仅让我们不再期待精确性，也让我们无法实现精确性。那些精确的系统试图让我们接受一个贫乏而规整的惨象——假装世间万物都是整齐地排列的。而事实上，现实是纷繁复杂的，天地间存在的事物也远远多于系统所设想的。要想获得大规模数据带来的好处，混乱应该是一种标准途径，而不应该是竭力避免的。

(3)　不是因果关系，而是相关关系。

在大数据时代，我们不必非得知道现象背后的原因，而是要让数据自己“发声”。通过给我们找到一个现象的良好关联物，相关关系可以帮助我们捕捉现在和预测未来。

在小数据世界中，相关关系也是有用的，但在大数据的背景下，相关关系大放异彩。通过应用相关关系，我们可以比以前更容易、更快捷、更清楚地分析事物。

大数据的相关关系分析法更准确、更快，而且不易受偏见的影响。建立在相关关系分析法基础上的预测是大数据的核心。

1.2　大数据的技术转型

直至今天，我们无论是使用 ATM 机取款、预订航班机票，还是查询交通违章信息，都离不开关系数据库，都依托于背后的关系数据库的数据事务处理。在事务处理上，关系模型无处不在。关系数据库模型成功的关键之处，在于该模型的标准化。每个数据单元在一个表中只会出现一次，这不但减少了冗余的存储成本，而且还使得数据修改只发生在一个位置，数据的一致性得以保持。表中的某一列可被指定为主键，主键是一个保证无二义性地检索单条记录的属性值，还可以在查询语法中使用主键来连接这些关系。关系数据库还创造出了结构化查询语言(Structured Query Language，SQL)来表达关系查询。关系型数据库为存储和查询数据提供了非常灵活的模型。

关系数据库模型的另一个重要概念是事务。事务管理器(或者一个事务处理服务)在一个工作单元中维护数据的完整性。一组操作被当作一个工作单元(unit)，在一个工作单元中，操作的所有部分一起成功，或失败并恢复。

凡符合关系模型的数据库，在事务处理上需要满足被称为 ACID 的特性。

- 原子性(Atomicity)：一个事务要被完全地、无二义性地做完或撤销。在任何操作出现一个错误的情况下，构成事务的所有操作的效果必须被撤销，数据应被回滚到事务开始前的状态。
- 一致性(Consistency)：一个事务应该保护所有定义在数据上的不变的属性(例如完整性约束)。在完成了一个成功的事务时，数据应处于一致的状态。一个一致的事务将保护定义在数据上的所有完整性约束。
- 隔离性(Isolation)：在同一个环境中，可能有多个事务并发执行，而每个事务都应表现为独立执行。串行地执行一系列事务的效果应该等同于并发地执行它们。在一个事务执行的过程中，数据的中间状态不应该被暴露给所有的其他事务；两个并发的事务应该不能操作同一项数据。
- 持久性(Durability)：一个被完成的事务的效果应该是持久的。只要事务成功结束，它对数据库所做的更新就必须永久保存下来。即使发生系统崩溃，重新启动数据库系统后，数据库还能恢复到事务成功结束时的状态。

关系数据库有两个强制性要求：对数据要预先理解，在进行插入操作之前，必须先定义好模式和关系；在写入操作发生后，保持数据的一致性。

可扩展性是这种计算模式的一大缺陷。当数据容量更大、并发处理性能需求更高时，唯有提高服务器性能指标和可靠性，这是典型的向上扩展模式(Scale Up)。即使可采用并行数据库集群，最多也只能管理有限数量的服务器，而且这种并行数据库也同样要求高配置的服务器才可以运转，其成本之高可以想象。

随着信息技术的进步，相比较而言，软件的重要性将下降，数据的重要性将上升。人类处理、传输和保存数据的能力不断增强。

20 多年来，CPU 的性能提高了 3500 倍；内存和磁盘的价格下降到了原来的 450 万分之一和 360 万分之一；主干网络带宽每 6 个月增加 1 倍，而每比特费用将趋于零。

另一方面，人类生产数据的能力也在增强，数据正在发生井喷式的增长。这与互联网、移动互联网、社交网络以及未来物联网大潮的高速发展直接相关。特别是智能手机等手持设备和社交网络的广泛使用，使得越来越多的人能够将可支配时间投入到各种应用中，而未来物联网的发展，任意物品和设施都有可能 24 小时不间断地产生状态数据。

让我们看一看当今互联网应用场景：Facebook 管理了超过 400 亿张图片，所需存储空间超过 100PB，每天发布的新消息超过 60 亿条，所需的存储空间超过 10TB；Twitter 一天产生 1.9 亿条微博；搜索引擎一天产生的日志高达 35TB，Google 一天处理的数据量超过 25PB；YouTube 一天上传的视频总时长为 5 万小时……。

数据量的衡量单位，从小到大依次为 KB、MB、GB、TB、PB、EB 和 ZB，相互之间的转换公式为 1024KB=1MB；1024MB=1GB；1024GB=1TB；1024TB=1PB；1024PB=1EB；1024EB=1ZB。我们曾经经历过 MB 存储时代、GB 存储时代，随着 IT 技术的快速发展，我们迈入了 TB 时代，而现在正向 PB、EB 时代迁移。

尽管随着时间的推移，商用计算机硬件变得越来越便宜，但是，从历史和经济的角度来看，持续不断地升级到更高配置的服务器硬件是不可行的。花费数倍的价钱升级一个大型机器，可能无法提供同样倍数的性能。相比之下，性能一般的小型服务器仍然很便宜。一般情况下，从经济的角度来看，水平扩展更有意义。换句话说，应该简单地为系统增加更多便宜的机器，而不是试图将一个关系型数据库放到一台昂贵的大型服务器上。

以关系数据库技术，不可能支撑今天大数据的应用场景。

对于很多应用场景，尤其是互联网相关应用来说，并不像银行业务等对数据的一致性有很高的要求，而更看重数据的高可用性以及架构的可扩展性等技术因素。因此，NoSQL 数据库应运而生，作为适应不同应用场景要求的新型数据存储与处理架构，它与传统数据库有很强的互补作用，而且应用场景更加广泛。例如，Yahoo 公司通过部署包含 4000 台普通服务器的 Hadoop 集群，可以存储和处理高达 4PB 的数据，整个分布式架构具有非常强的可扩展性。NoSQL 数据库的广泛使用，代表了一种技术范型的转换。

1.3 数据分片

在大数据环境下，数据量已经由 GB 级别跨越到 PB 级别，依靠单台计算机已经无法存储与处理如此规模的数据，唯一的出路，是采用大规模集群来对这些数据进行存储和处理，所以，系统的可扩展性成为衡量系统优劣的关键因素。

传统关系数据库系统为了支持更多的数据，采用纵向扩展(Scale Up)的方式，即不增加机器数量，而是通过改善单机硬件资源配置，来解决问题。如今这种方式已经行不通了。

目前主流的大数据存储与计算系统通常采用横向扩展(Scale Out)的方式支持系统可扩展性，即通过增加机器数目来获得水平扩展能力。与此对应，对于待存储处理的海量数据，需要通过数据分片(Shard/Partition)来对数据进行切分并分配到各个机器中去，通过数据分片

实现系统的水平扩展。

目前，大规模存储与计算系统都是采用普通商用服务器来作为硬件资源池的，系统故障被认为是常态。因此，与数据分片密切相关的是数据复制，通过数据复制来保证数据的高可用性。数据复制是将同一份数据复制存储在多台计算机中，以保证数据在故障常发环境下仍然可用。从数据复制还可以获得另一个好处，即可以增加读操作的效率，客户端可以从多个备份数据中选择物理距离较近的进行读取，既增加了读操作的并发性，又可以提高单次的读取效率。

数据复制带来的难题是如何保证数据的一致性。由于每份数据存在多个副本，在并发地对数据进行更新时，如何保证数据的一致性就成为关键问题。

可以将数据分片的通用模型看作是一个二级映射关系。第一级映射是 key-partition 映射，即把数据记录映射到数据分片空间，通常，一个数据分片包含多条记录数据；第二级映射是 partition-machine 映射，把数据分片映射到物理机器中，即一台物理机器通常可以容纳多个数据分片。

在做数据分片时，根据 key-partition 关系，将数据水平切割成众多数据分片，然后再按照 partition-machine 映射关系，将数据分片存储到对应的物理机器上。而在做数据访问时，比如要查找某条记录的值 Get(key)，首先根据 key-partition 映射找到对应的数据分片，然后再查找 partition-machine 关系表，就可以找到哪台物理机器存储该条数据，之后，即可从相应的物理机器读取 key 对应的 value 内容了。

数据分片有两种常用策略：哈希分片与范围分片。对于哈希分片来说，因为其主要通过哈希函数来建立 key-partition 映射关系，因此，哈希分片只支持"单点查询"(Point Query)，即根据某个记录的主键(key)获得记录内容，而无法支持"范围查询"(Range Query)，即指定记录的主键范围，一次读取多条满足条件的记录。采取哈希分片的实际系统众多，大多数 KV(key-value，键值)存储系统都支持这种方式。

与此相对应地，范围分片的系统则既可以支持单点查询，也可以支持范围查询。范围分片首先对所有记录的主键进行排序，然后在排好序的主键空间里将记录划分成数据分片，每个数据分片存储有序的主键空间片段内的所有记录。

1.4　数据一致性

在大数据系统中，为了获得系统可用性，需要为同一数据分片存储多份副本，业界的常规做法是一个数据分片同时保存三个副本。将数据复制成多份除了能增加存储系统的可用性，同时还能增加读操作的并发性，但引发了数据一致性问题，即同一数据分片存在多个副本。在并发的写请求下，如何保持数据一致性尤为重要，即在存储系统外部的使用者看来，即使存在多个副本数据，它与单份数据也应该是一样的。

CAP、BASE、ACID 等基本原则是分布式环境下数据一致性方案设计重要的指导原则。

1.4.1　CAP 原则

CAP 是对 Consistency/Availability/Partition Tolerance 的一种简称，CAP 原则对分布式系

统中的三个特性进行了如下归纳。

- 一致性(Consistency)：在分布式系统中的所有数据备份，在同一时刻是否具有同样的值(等同于所有节点访问同一份最新的数据副本)。
- 可用性(Availability)：在集群中，一部分节点出现故障后，集群整体是否还能响应客户端的读写请求(对数据更新具备高可用性)。
- 分区容错性(Partition Tolerance)：从实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在 C 和 A 之间做出选择。

CAP 原则是指对于一个大规模分布式数据系统来说，CAP 三个特性不可兼得，同一个系统至多只能实现其中的两个，而必须放宽第三个要素来保证其他两个要素被满足。即要么 AP，要么 CP，抑或 AC，但是不存在 CAP，如图 1-1 所示。

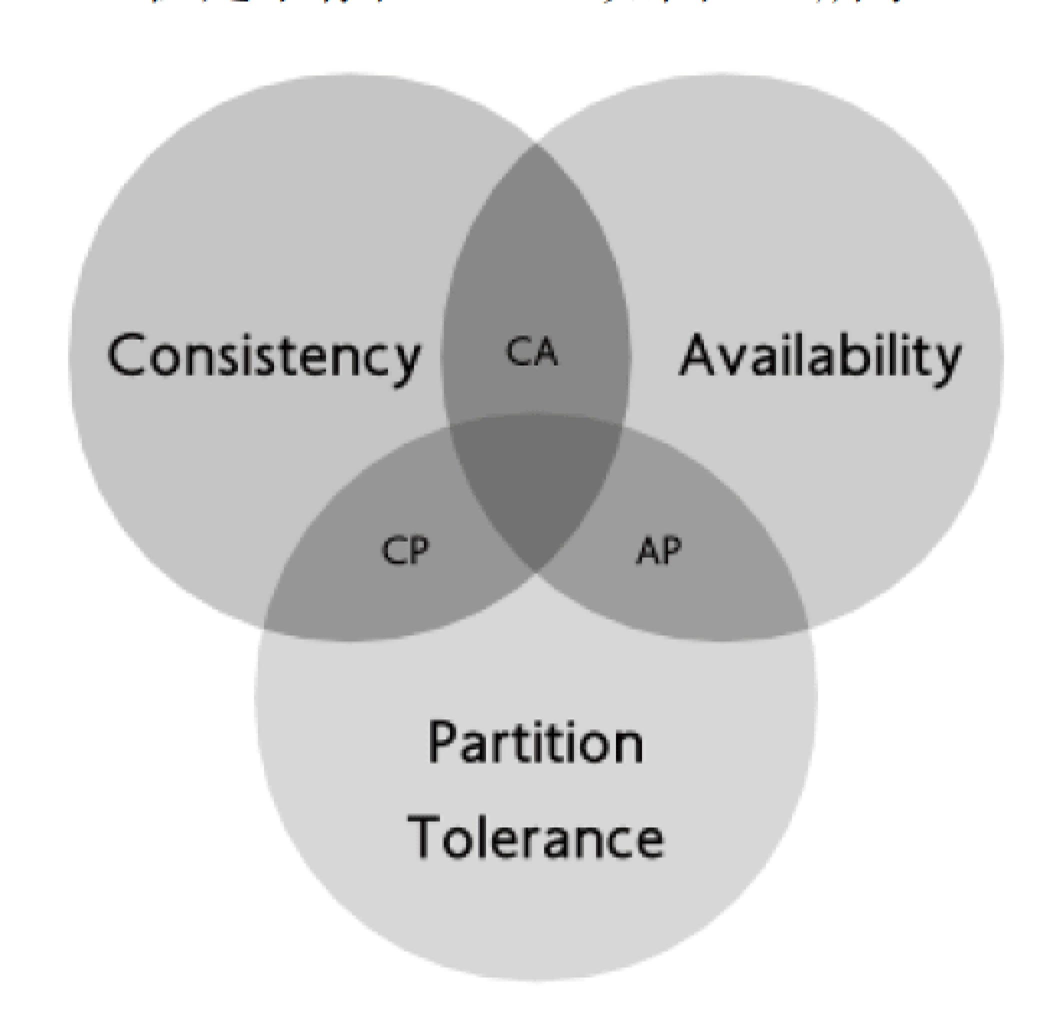


图 1-1　CAP 原则示意

我们可以认为，在网络环境下，运行环境出现网络分区是不可避免的，所以系统必须具备分区容忍性特性，于是，一般在此种场景下，设计大规模分布式系统时，架构师往往在 AP 和 CP 中进行权衡和选择，有所强调、有所放弃。

在一个分布式系统中，如果数据无副本，那么系统首先就满足强一致性条件，单一副本数据，不可能发生数据不一致的情况。此时，系统具备 C，如果我们容忍分区容错性 P，意味着系统可能发生网络分区状况，如有宕机现象出现时，就必然导致某些数据不可访问，此时，可用性 A 是不能被满足的，即在此情形下获得了 CP 特性，但 CAP 不可同时满足。

如果系统中数据有副本，假设一份数据有分别存储在不同机器上的两个副本，最初数据是保持一致的，某一时刻，机器 1 上对这个数据进行了更新操作，这个更新操作随后会同步到机器 2 上，使两个副本保持一致性。但网络分区是不可忽视的，可以设想，在数据复制同步未完成的情况下，发生了网络分区，导致两台机器无法通信，这时，我们就不得不在 C 或 A 之间做权衡和选择。如果希望系统可用性优先(选择 A)，那么对于读取机器 2 上的数据查询请求返回的并非是最新的数据，此种选择就放弃一致性(C)，产生数据不一致情况。如果选择一致性优先(选择 C)，那么，在两台机器恢复通信并将数据同步到一致状态

前，对于机器 2 就要拒绝对数据的读请求，此时，可用性无法保证(放弃 A)。所以不论选择哪一个，必然以牺牲另外一个因素作为代价，也就是说，要么 AP，要么 CP，但不会有 CAP。

对于分布式系统来说，分区容错性是天然具备的，所以在设计具体分布式架构技术方案时，只能对一致性和可用性两个特性做出取舍，要么选择强一致性，减弱服务可用性，要么选择高可用性而容忍弱一致性。

我们可以回顾一下传统关系数据库的 ACID 特性，在 CAP 三要素中，传统关系数据库选择 CA 两个特性，即强一致性、高可用性，与分区容错性这个天然要素不可调和，这是造成其在分布式环境下可扩展性差的根本原因。而 NoSQL 系统则更关注 AP 因素，即高可用性和分区容错性，也意味着兼顾高可用性和高可扩展性，而牺牲强一致性。对于绝大多数互联网应用来说，高可用性直接涉及用户体验，而对数据一致性要求并不高，NoSQL 系统这类以弱一致性作为代价的系统，正是适应了这一现实需求。

网络分区(P)虽然是个天然属性，但在现代的实际系统中，依然是个小概率事件。以此为出发点，我们并不应该为了容忍这种小概率事件而在设计之初就选择放弃 A 或者放弃 C，在大多数没有出现网络分区的状况下，还是可以尽可能兼顾 AC 两者，即有条件地兼顾 CAP 三要素。

另一个需要说明的是，即使必须在 AC 之间做出取舍，也不应该是粗粒度地在整个系统级别进行取舍，即整个系统要么取 A 舍 C，要么取 C 舍 A，而是应该考虑系统中存在不同的子系统，甚至应该在不同的系统运行时或者在不同的数据间进行灵活的差异化的细粒度取舍，即可能对不同子系统采取不同的取舍策略，在不同的系统运行时，对不同数据采取不同的取舍策略。因此，CAP 三者并非是绝对的两要素有或没有，可以看作是在一定程度上的有或没有，考虑的是在多大程度上进行取舍。

由此可以看出，CAP 的修改策略是可取的。在绝大多数系统未产生网络分区的情形下，应该尽可能保证 AC 两者兼得，也即大多数情况下，考虑 CAP 三者兼得，当发生网络分区时，系统应该能够识别这种状况并对其进行正确处理，在网络分区场景下进入明确的分区模式，此时，可能会限制某些系统操作，最后，在网络分区解决后，能够进行善后处理，即恢复数据的一致性，或者弥补分区模式中产生的错误。

1.4.2 CAP 与 ACID

谈到 CAP 与 ACID 之间存在的关系，首先因为 CAP 和 ACID 两者都包含了 A 和 C，但是其具体含义有所不同；其次，如果 CAP 中选择 A 的话，在一定程度上，会影响 ACID 中的部分要求。

CAP 与 ACID 两者中尽管都包含一致性，但是，两者的含义不同，ACID 中的 C 指的是对操作的一致性约束，而 CAP 中的 C 指的是数据的强一致性(多副本对外表现类似于单副本)，所以，可以将 CAP 中的 C 看作一致性约束的一种，即 CAP 中的 C 是 ACID 中的 C 所涵盖语义的子集。在出现网络分区的情形下，ACID 中的 C 所要求的一致性约束是无法保证的，所以，在网络分区解决后，需要通过一定手段来恢复 ACID 中要求的一致性。

当出现网络分区时，ACID 中的事务独立只能在多个分区中的某个分区执行，因为事务的序列化要求通信，而当网络分区时，明显无法做到这点，所以只能在某个分区执行。如果多个分区都可以各自进行 ACID 中的数据持久化(D)操作，当网络分区解决后，如果每个

分区都提供持久化记录，则系统可以根据这些记录发现违反 ACID 一致性约束的内容，并给予修正。

1.4.3 BASE 原则

关系数据库系统采纳 ACID 原则，获得高可靠性和强一致性。而大多数分布式环境下的云存储系统和 NoSQL 系统则采纳 BASE 原则。BASE 原则具体是指如下几项。

- 基本可用(Basically Available)：在绝大多数时间内，系统处于可用状态，允许偶尔的失效，所以称为基本可用。
- 软状态或者柔性状态(Soft State)：是指数据状态不要求在任意时刻都完全保持同步，可以理解为系统处于有状态(State)和无状态(Stateless)之间的中间状态。
- 最终一致性(Eventual Consistency)：与强一致性相比，最终一致性是一种弱一致性，尽管软状态不要求任意时刻数据保持一致同步，但是，在给定时间窗口内，最终会达到一致的状态。

BASE 原则与 ACID 原则有很大的差异。BASE 通过牺牲强一致性来获得高可用性。尽管现在大多数的 NoSQL 系统采纳了 BASE 原则，但是有一点值得注意：NoSQL 系统与云存储系统的发展过程正在向逐步提供局部 ACID 特性发展，即从全局而言，符合 BASE 原则，但局部上支持 ACID 原则，这样，就可以吸取两者各自的好处，在两者之间建立平衡。

ACID 强调数据的一致性，这是传统数据库设计的思路。而 BASE 更强调可用性，弱化数据强一致性的概念，这是互联网时代对于大规模分布式数据系统的一种需求，尤其是其中的软状态和最终一致性。可以说，ACID 和 BASE 原则是在明确提出 CAP 理论之前关于如何对待可用性和强一致性的两种完全不同的设计思路。

1.5 主流大数据技术

目前，主流的开源大数据技术的基础原理皆基于上述基本理论。主流的大数据技术可以分为两大类。

一类面向非实时批处理业务场景，着重用于处理传统数据处理技术在有限的时空环境里无法胜任的 TB 级、PB 级海量数据存储、加工、分析、应用等。一些典型的业务场景如：用户行为分析、订单防欺诈分析、用户流失分析、数据仓库等，这类业务场景的特点，是非实时响应，通常，一些单位在晚上交易结束时，抽取各类数据进入大数据分析平台，在数小时内获得计算结果，并用于第二天的业务。比较主流的支撑技术为 HDFS、MapReduce、Hive 等。

另一类面向实时处理业务场景，如微博应用、实时社交、实时订单处理等，这类业务场景，特点是强实时响应，用户发出一条业务请求，在数秒钟之内要给予响应，并且确保数据完整性。比较主流的支撑技术为 HBase、Kafka、Storm 等。

这里先简要介绍这些技术的特点，针对这些技术的详细使用，在本书后面会安排。

(1) HDFS。

HDFS 是 Hadoop 的核心子项目，是整个 Hadoop 平台数据存储与访问的基础，在此之

上，承载其他如 MapReduce、HBase 等子项目的运转。它是易于使用和管理的分布式文件系统。

HDFS 是一个高度容错性的系统，适合部署在廉价的机器上。HDFS 能提供高吞吐量的数据访问，非常适合大规模数据集上的应用。HDFS 放宽了一部分 POSIX 约束，来实现流式读取文件系统数据的目的。

(2) MapReduce。

MapReduce 是一个软件架构，在数以千计的普通硬件构成的集群中以平行计算的方式处理海量数据，该计算框架具有很高的稳定性和容错能力。MapReduce 对负责逻辑进行高度归约，抽象为 Mapper 和 Reducer 类，复杂逻辑通过理解，转化为符合 MapReduce 函数处理的模式。

MapReduce job 会划分输入数据集为独立的计算块，这些分块被 map 任务以完全并行、独立的模式处理。MapReduce 框架对 maps 的输出进行排序，排序后，数据作为 reduce 任务的输入数据。job 的 input 和 output 数据都存储在 HDFS 文件系统中。计算框架管理作业调度、监控作业、重新执行失败任务。

(3) YARN。

Apache Hadoop YARN(Yet Another Resource Negotiator，另一种资源协调者)是从 Hadoop 0.23 进化来的一种新的资源管理和应用调度框架。基于 YARN，可以运行多种类型的应用程序，例如 MapReduce、Spark、Storm 等。YARN 不再具体管理应用，资源管理和应用管理是两个低耦合的模块。

YARN 从某种意义上来说，是一个云操作系统(Cloud OS)。基于该操作系统之上，程序员可以开发多种应用程序，例如批处理 MapReduce 程序、Spark 程序以及流式作业 Storm 程序等。这些应用，可以同时利用 Hadoop 集群的数据资源和计算资源。

(4) HBase。

HBase 是 Hadoop 平台中重要的非关系型数据库，它通过线性可扩展部署，可以支撑 PB 级数据存储与处理能力。

作为非关系型数据库，HBase 适合于非结构化数据存储，它的存储模式是基于列的。

(5) Hive。

Hive 是 Apache 基金会下面的开源框架，是基于 Hadoop 的数据仓库工具，它可以把结构化的数据文件映射为一张数据仓库表，并提供简单的 SQL(Structured Query Language)查询功能，后台将 SQL 语句转换为 MapReduce 任务来运行。

使用 Hive，可以满足一些不懂 MapReduce 但懂 SQL 的数据库管理员的需求，让他们能够平滑地使用大数据分析平台。

(6) Kafka。

Apache Kafka 是分布式“发布-订阅”消息系统，最初，它由 LinkedIn 公司开发，而后成为 Apache 项目。Kafka 是一种快速、可扩展的、设计时内在地就是分布式的、分区的和可复制的提交日志服务。

Kafka 是一个分布式系统，易于向外扩展，可为发布和订阅提供高吞吐量，并且支持多订阅者，当失败时，能自动平衡消费者；Kafka 可将消息持久化存储，既可面向非实时业务，也可以面向实时业务。

(7) Storm。

Storm 是一个免费开源、分布式、高容错的实时计算系统。它能够处理持续不断的流计算任务，目前，比较多地被应用到实时分析、在线机器学习、ETL 等领域。

1.6 大数据职业方向

大数据作为一种趋势，已吸引了越来越多的重视，目前，上至国家部委、下至普通公司，已纷纷开展各类大数据平台建设与分析应用，这无疑对想要从事大数据方面工作的 IT 人员提供了难得的历史机遇。

本书的主要定位是大数据实践入门，通过对主流大数据技术的覆盖性讲解，以及动手实践，帮助从业者快速入门。入门之后，可以根据个人兴趣以及职业趋势，选择适合自己的发展方向。

目前就本书看来，大数据技术方面比较有前景的就业方向主要有如下几类。

一是大数据平台架构与研发：从事底层的大数据平台研发，这类方向技术难度最高，适合于前沿技术机构，要不断发现与改进目前大数据技术的缺陷，对于形成稳定版本的大数据平台，要面向业界进行推广。这类岗位整体数量不多，但方向会比较专注。

二是大数据平台应用开发：从事大数据平台应用技术的开发工作，满足大量企事业单位使用大数据平台的需求，这类岗位会比较充足，需要不断学习各类大数据平台，并应用到开发项目中。

三是大数据平台集成与运维：从事大数据平台的集成与运维工作。对于大量的企事业单位的大数据部署与常规应用来说，需要有专职的集成人员进行集成安装与调试，需要定期运维人员进行运维与提供技术保障，这类岗位也会比较充足，但需要熟练掌握大数据平台的使用，针对问题能够及时解决。

四是大数据平台数据分析与应用：从事数据分析、预测与应用工作，借助于大数据平台，分析各类业务数据，并服务于业务，这类岗位跟业务休戚相关，一些对数据高度重视的机构，如精确广告营销、大数据安全分析等单位，对此会有充足的人才需求。

五是大数据技术培训与推广：从事大数据技术教育与培训工作。这类工作机关会随着大数据人才需求热度的提高而不断增加岗位人数，与此同时，随着推广程度的延伸，也会促生更多的机构，会有更多的人才加入大数据领域。

1.7 大数据实践平台的搭建

针对大数据技术的学习，本书假定读者已掌握 Linux 的基本使用、Java 简单编程。

本书的定位是实践，所以在进入后面的学习前，需要动手搭建实践环境。这里给出几种搭建模式，帮助读者建立适合自己的环境。

1.7.1 初学者模式

本模式是在一台物理机(笔记本电脑或台式机)上，根据计算机硬件性能，搭建一台或三

台虚拟主机，在虚拟主机里部署各类大数据组件。

(1) 所需要的主要软件环境如下。

- 虚拟主机软件：推荐采用 VMware Workstation 或 Oracle VM VirtualBox。
- Linux 操作系统：推荐采用 Red Hat 或 CentOS。
- 远程管理软件：SSH 或 SecureCRT。

(2) 本模式的安装流程如下。

① 在物理机上安装虚拟主机软件。

② 在虚拟主机软件中新建虚拟主机，安装 Linux。

③ 调通新建虚拟主机的网络，可以实现在物理机与虚拟主机之间互访问。

④ 开启 SSH 服务，实现物理主机通过 SSH 管理虚拟主机。

本模式适合初学者，优点是简单，缺点是每次使用大数据时，需要先开启虚拟主机，无法随时随地使用。

1.7.2 物理集群模式

本模式是采用多台(1 至 3 台)物理服务器，在标准的机房环境里搭建大数据平台。

(1) 所需要的主要软硬件环境如下：

- 多台物理服务器。
- 一台交换机。
- 个人管理机，笔记本电脑或台式机。
- Linux 操作系统，推荐采用 Red Hat 或 CentOS。
- 远程管理软件，SSH 或 SecureCRT。

(2) 本模式的安装流程如下。

① 服务器上架与物理连线。

② 在所有物理服务器上安装 Linux 操作系统。

③ 调通物理服务器之间的网络，调通管理机对物理服务器的网络。

④ 在所有物理服务器上开启 SSH 服务，并增加机房网络防火墙安全策略，允许管理机通过 SSH 管理物理服务器。

本模式适合于机构用户，有一定的硬件资源，优点是性能优越，可永备使用；缺点是成本相对较高，集群规模有限。

1.7.3 虚拟化集群模式

本模式是采用云计算的模式，底层设置多台物理服务器，通过云计算管理软件实现几十、上百台虚拟主机的制备，在虚拟主机中搭建大数据平台。

(1) 所需要的主要软硬件环境如下：

- 一定数量的物理服务器。
- 一定数量的交换机。
- 个人管理机，笔记本电脑或台式机。
- OpenStack 等云管理软件。

- Linux 操作系统：推荐采用 Red Hat 或 CentOS。
- 远程管理软件：SSH 或 SecureCRT。

(2) 本模式的安装流程如下。

① 服务器上架与物理连线。

② 在所有物理服务器上安装 Linux 操作系统。

③ 部署 OpenStack 框架，在物理服务器上按要求部署 Nova、Ceph、Glance、Keystone、Quantum、MySQL、HTTP、Horizon 等组件模块，完成云计算环境的部署。

④ 调通物理服务器、云计算内部网络，调通管理机对物理服务器的网络。

⑤ 制备 Linux 虚拟主机，并开放相关的 SSH 管理权限。

本模式适合于中型以上机构用户，利用富余的硬件资源搭建云环境，并按需对外提供虚拟主机资源，优点是可在分钟级内批量创建或回收上百个大数据节点，满足更多用户、更大范围的使用；缺点是需要维护云计算环境，如果规模不大，成本会比较高。

1.8 小　　结

本章主要介绍分布式系统数据处理的基本原则，CAP、BASE 等，以及与传统关系数据库 ACID 原则的关系，为读者理解和学习后续章节介绍的平台和工具提供预备知识。

第 2 章

HDFS 文件系统

学习目标

本章介绍 Hadoop 的核心组成部分 HDFS 文件系统，包括其原理、安装与配置、管理及外部编程接口等。通过对本章内容的学习，使读者掌握分布式文件系统的主要结构、HDFS 文件系统的内部运行原理和机制、HDFS 的数据读写方式，同时，了解 HDFS 文件系统的数据传输和存储模式。

本章最后将详细介绍 Hadoop 的安装和基本配置。学习完本章后，读者可以搭建自己的 Hadoop 集群。

本章要点

- HDFS 文件系统的结构与组成
- HDFS 系统的数据读写
- HDFS 系统的数据存储及数据完整性
- Hadoop 的安装及配置

2.1 HDFS 概述

Hadoop 实现了一个分布式文件系统(Hadoop Distributed File System，HDFS)，HDFS 是 Apache Hadoop Core 项目的一部分，是 Hadoop 兼容性最好的标准级分布式文件系统。

2.1.1 分布式文件系统

当今的信息时代中，人们可以获取的数据成指数倍地增长。单纯通过增加硬盘个数来扩展计算机文件系统的存储容量的方式，在容量大小、容量增长速度、数据备份、数据安全等方面都不适用，对于数据量很大的应用系统来说尤其如此。分布式文件系统可以有效解决数据的存储和管理难题。

分布式文件系统(Distributed File System，DFS)指通过一套管理系统，能够将文件分散至不同的计算机进行存储，并通过规范的标准协议，方便客户机进行高效存取。

与单机的文件系统不同，分布式文件系统不是将数据放在一块磁盘上由上层操作系统来管理，而是存放在一个服务器集群上，由集群中的服务器通过各尽其责、通力合作的方式提供整个文件系统的服务。将固定于某个地点的某个文件系统，扩展到任意多个地点/多个文件系统，这些节点组成一个文件系统网络。每个节点可以分布在不同的地点，通过网络进行节点间的通信和数据传输。人们在使用分布式文件系统时，无须关心数据是存储在哪个节点上，或者是从哪个节点获取的，只需要像使用本地文件系统一样管理和存储文件系统中的数据即可。

分布式文件系统中，重要的服务器包括：主控服务器(Master/NameNode)、数据服务器(一般称为 ChunkServer 或 DataNode)和客户服务器(Client)。分布式文件系统的典型架构如图 2-1 所示。

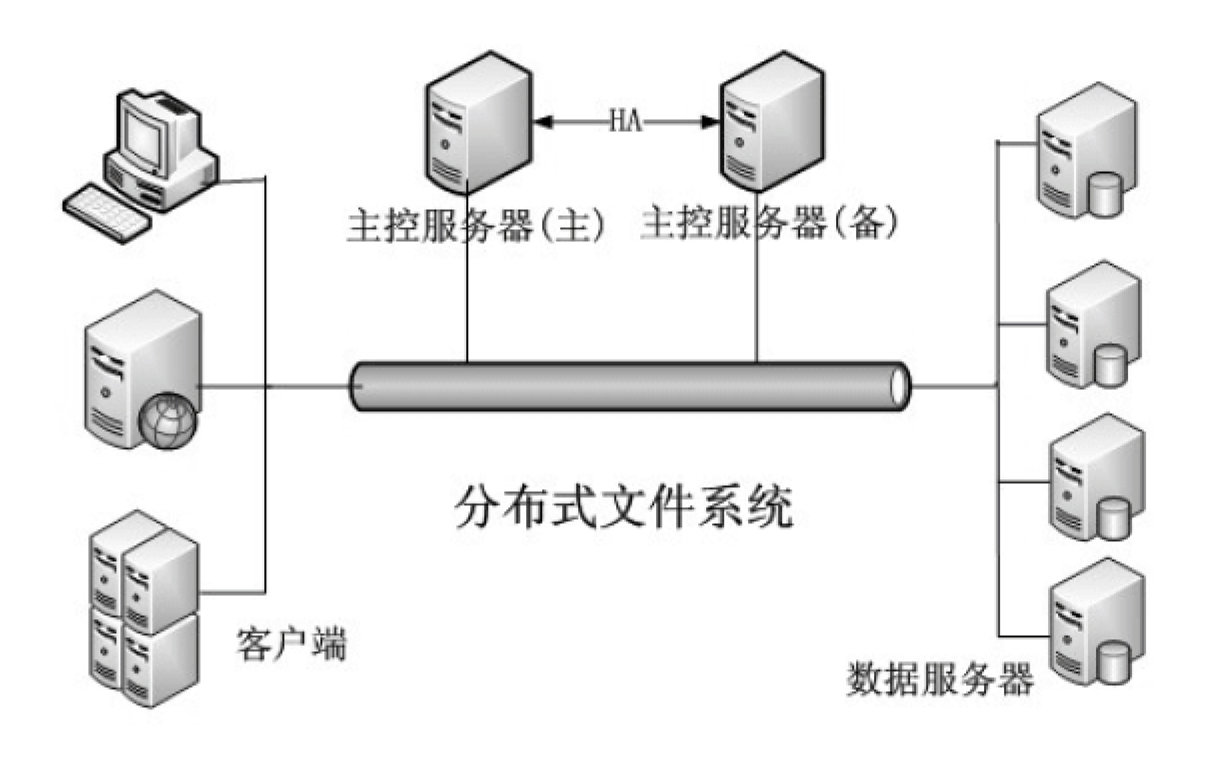


图 2-1 典型分布式文件系统的结构

1. 分布式文件系统的特点

与传统文件系统相比，分布式文件系统具有以下主要特点。

(1) 可扩展性强。扩展能力是一个分布式文件系统最重要的特点。基本上，所有的分布式文件系统都支持随时随地对数据服务器进行扩展，提升存储容量和访问带宽等。有的

系统还支持多个目录/主控服务器。

(2) 统一命名空间。采用统一命名空间，分布式文件系统对于客户端是完全透明的，客户端看到的是统一的全局命名空间，用户操作起来就像是管理本地文件系统。通过元数据管理，文件以块的方式采用多副本模式进行存放。

(3) 高性能。由于一个文件被分成多份，保存在不同的数据服务器上，访问时，可以同时读取，性能会达到最优。

(4) 高可用性。分布式文件系统必须具有高容错能力，即无论是客户端还是服务器出现故障，都不会影响整个系统的功能。为了做到这一点，单点失效是必须被避免的，例如使用资源冗余技术或者提供失效恢复服务。单个数据节点的故障并不会影响集群整体运转。

(5) 弹性存储。可以根据业务需要灵活地增加或缩减数据存储以及增删存储池中的资源，而不需要中断系统运行。弹性存储的最大挑战，是减小或增加资源时的数据震荡问题。

2. 常见的分布式文件系统

分布式文件系统既有开源软件平台解决方案，如 Hadoop HDFS、Fast DFS 等；也有非开源平台解决方案，如最为著名的 Google FS、也有像 Windows Server 2003/2008 平台上的 DFS 组件等。

分布式文件系统在当前应用普遍，产品种类丰富。下面介绍几种典型的系统。

(1) Lustre。

Lustre 最早是由 HP、Cluster File System 联合美国能源部共同开发的 Linux 平台下的分布式集群文件系统，后期由于 Cluster File System 公司被 Sun 收购，而 Sun 又被 Oracle 收购，因此，Lustre 官方网站目前挂靠在 Oracle 公司(http://wiki.lustre.org/index.php/Main_Page)。

Lustre 主要面向超级计算机，拥有超强可扩展性与可靠性，能够支持上万个节点、PB 级存储、100GB/s 的高速访问能力。

Lustre 采用 GPL 许可协议，属于开放源代码的分布式集群文件系统，开发语言采用 C/C++，使用平台为 Linux；当前，除了 Oracle 公司外，有新成立的名为 Whamcloud 的公司专注于 Lustre 平台的开源研发，其官方网站为 http://www.whamcloud.com/。

(2) Google FS。

Google FS(Google File System)是谷歌公司开发的一个分布式可扩展的文件系统，它主要用于大型、分布式、大数据量的互联网应用平台。

Google FS 被设计运行在廉价普通的 PC 服务器上，提供多数据副本实现数据冗余，通过数据分块并行存取，满足互联网用户的海量数据存储需求。

Google FS 最早是由 Google 工程师于 2003 年发表的一篇学术文章 The Google File System 而为世人所熟知的，Google FS 提供了相似的访问接口，如 read、write、create、delete、close 等，使得开发者可以非常方便地使用。

Google FS 运行于 Linux 平台上，开发语言是 C/C++，本身并不开源，本章中所介绍的 Hadoop 平台，是在受到 Google FS 启发后，采用其理念重新用 Java 语言实现的一个开源平台。

(3) Fast DFS。

Fast DFS 是一个类 Google FS 的开源分布式文件系统，它由 C/C++语言开发，可运行于

Linux、Unix、AIX 平台。Fast DFS 提供专用文件存取访问方式，不支持 POSIX 接口方式，在系统中也不能使用 mount 方式挂接。FastDFS 在架构上充分考虑了冗余备份、负载均衡、可扩展等问题，平台本身具有高可用、高性能等优点。Fast DFS 支持文件的高效存储、同步、上传、下载等，比较适合于互联网视频网站、文档分享网站、图片分享网站等应用。

2.1.2 HDFS 介绍

HDFS 是 Hadoop 的核心子项目，是整个 Hadoop 平台数据存储与访问的基础，在此之上，承载其他如 MapReduce、HBase 等子项目的运转。

HDFS 是类似于 Google FS 的开源分布式文件系统，被设计成适合运行在通用硬件上的分布式文件系统。它与现有的分布式文件系统有很多共同点。但同时，它与其他的分布式文件系统的区别也是很明显的。

HDFS 是一个高度容错性的系统，适合部署在廉价的机器上。HDFS 能提供高吞吐量的数据访问，非常适合大规模数据集上的应用。HDFS 放宽了一部分 POSIX 约束，来实现流式读取文件系统数据的目的。

HDFS 是易于使用与管理的分布式文件系统，主要特点和设计目标如下。

1. 硬件故障是常态

整个 HDFS 系统可以由数百或数千个存储着文件数据片段的服务器组成。实际上，它里面有非常巨大的组成部分，每一个组成部分都很可能出现故障，这就意味着 HDFS 里总是有一些部件是失效的，因此故障的检测和自动快速恢复是 HDFS 一个很核心的设计目标。

2. 流式数据访问

HDFS 被设计成适合批量处理的，而不是用户交互式的。POSIX 的很多硬性需求对于 HDFS 应用都是非必需的，HDFS 放宽了 POSIX 的要求，这样，可以实现以流的形式访问(Streaming Access)文件系统中的数据。同时去掉 POSIX 一小部分关键语义，可以获得更好的数据吞吐率。

3. 简单的一致性模型

大部分 HDFS 程序对文件操作需要的是一次写、多次读取的操作模式。HDFS 假定一个文件一旦创建、写入、关闭之后就不需要修改了。这简单化了数据一致的问题，并使高吞吐量的数据访问变得可能。

4. 名字节点(NameNode)和数据节点(DataNode)

HDFS 是一个主从结构，一个 HDFS 集群包括一个名字节点(也叫名称节点)，它是一个管理文件命名空间和调节客户端访问文件的主服务器，当然，还有一些数据节点，通常是一个节点一个机器，它来管理对应节点的存储。HDFS 对外开放文件命名空间，并允许用户数据以文件形式存储。内部机制是将一个文件分割成一个或多个块，这些块被存储在一组数据节点中。名字节点用来操作文件命名空间的文件或目录操作，如打开、关闭、重命名等。它同时确定块与数据节点的映射。数据节点负责来自文件系统客户户的读写请求。数据节点同时还要执行块的创建、删除，以及来自名字节点的块复制指令。

5. 大规模数据集

HDFS 被设计为 PB 级以上存储能力，单个的存储文件可以是 GB 或者 TB 级。因此，HDFS 的一个设计原则是支持成千上万大数据文件的存储，即将单个文件分成若干标准数据块，分布存储于多个节点上，当用户访问整个文件时，由这些节点集群向用户传输所拥有的数据块，由此可以获得极高的并行数据传输速率。

6. 可移植性

HDFS 在设计之初，就考虑到了异构软硬件平台间的可移植性，能够适应于主流硬件平台。它基于跨操作系统平台的 Java 语言进行编写，这有助于 HDFS 平台的大规模应用推广。

名字节点是整个 HDFS 的核心。一个标准的 HDFS 集群应由名字节点、备用名字节点、数据节点组成，HDFS 的基本结构如图 2-2 所示。

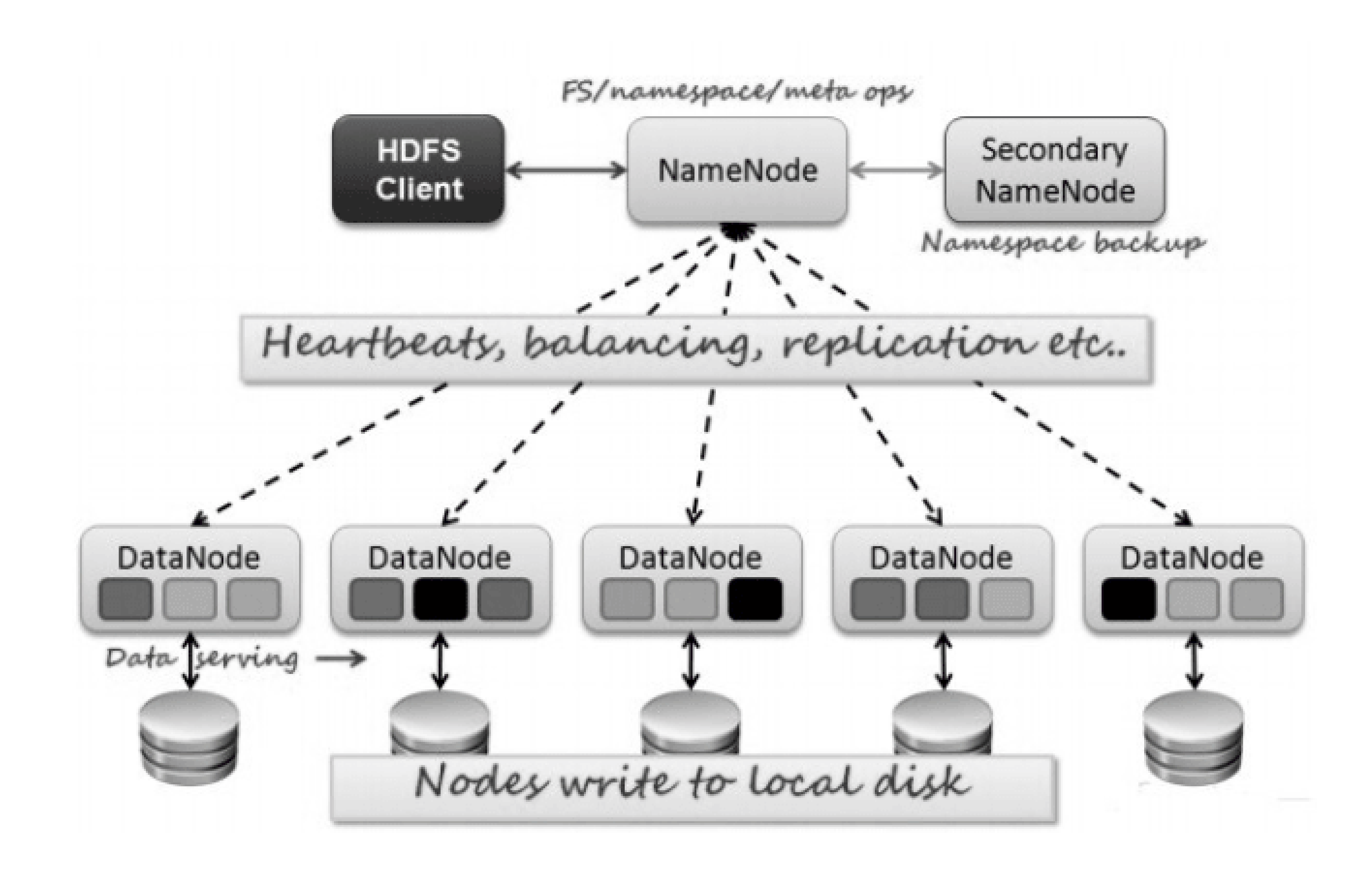


图 2-2　HDFS 系统的基本结构

集群中，一台机器上只运行一个 NameNode 实例，而集群中其他机器分别运行一个 DataNode 实例。NameNode 是一个中心服务器，负责管理文件系统的名字空间以及客户端对文件的访问，用户能够以文件的形式在上面进行名字空间操作，比如打开、关闭、重命名文件或目录，同时，NameNode 还决定了数据块到数据节点的映射关系。NameNode 也可以称为管理文件系统的元数据。集群中，每一个节点配置一个 DataNode，每个 DataNode 负责管理它所在节点上的数据存储。从内部看，一个文件被分成一个或多个数据块，这些块存储在一组 DataNode 上。同时，DataNode 负责处理文件系统客户端的读写请求，在 NameNode 的统一调度下进行数据块的创建、删除和复制。

HDFS 的数据块：磁盘存储文件时，是按照数据块(block)来存储的，也就是说，数据块是磁盘读/写的最小单位。数据块也称磁盘块。在 HDFS 中也有块的概念，默认为 64MB，每个块作为独立的存储单元。

基于数据块的存储方式非常适合用于备份，可提供数据容错能力和可用性(如图 2-3 所示)。HDFS 提供给应用程序例如 MapReduce 数据服务。一般来说，MapReduce 的 Map 任务通常一次处理一个块中的数据，如果任务数太少(少于集群中节点的数量)，就没有发挥多节点的优势，甚至作业的运行速度就会与单节点一样。

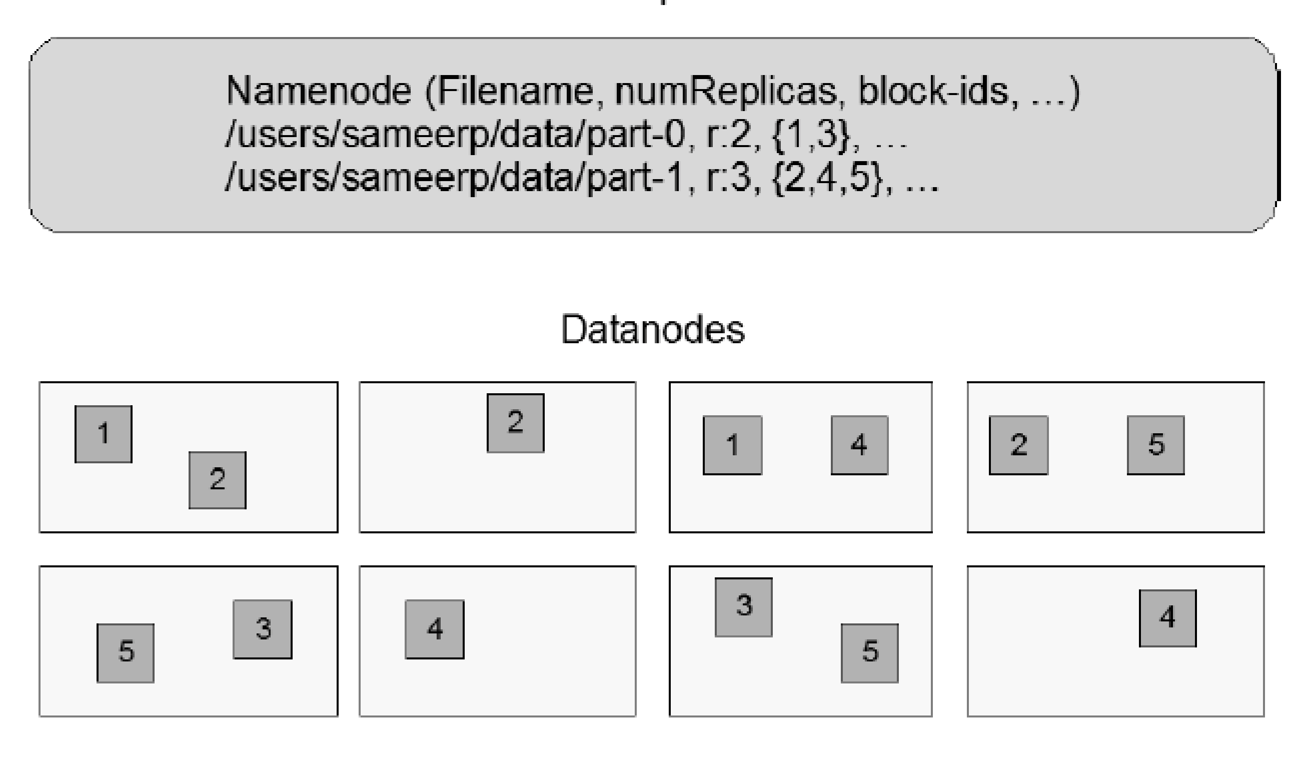


图 2-3　HDFS 块副本

2.2　HDFS 的运行机制

本节将详细介绍 HDFS 的结构与运行原理。

2.2.1　HDFS 的结构与组成

HDFS 采用主/从(Master/Slave)结构，整个集群由一个名字节点和多个数据节点组成。

NameNode 主要负责管理文件命名空间和客户端访问的主服务器，而 DataNode 则负责对存储进行管理。

HDFS 的体系结构如图 2-4 所示。

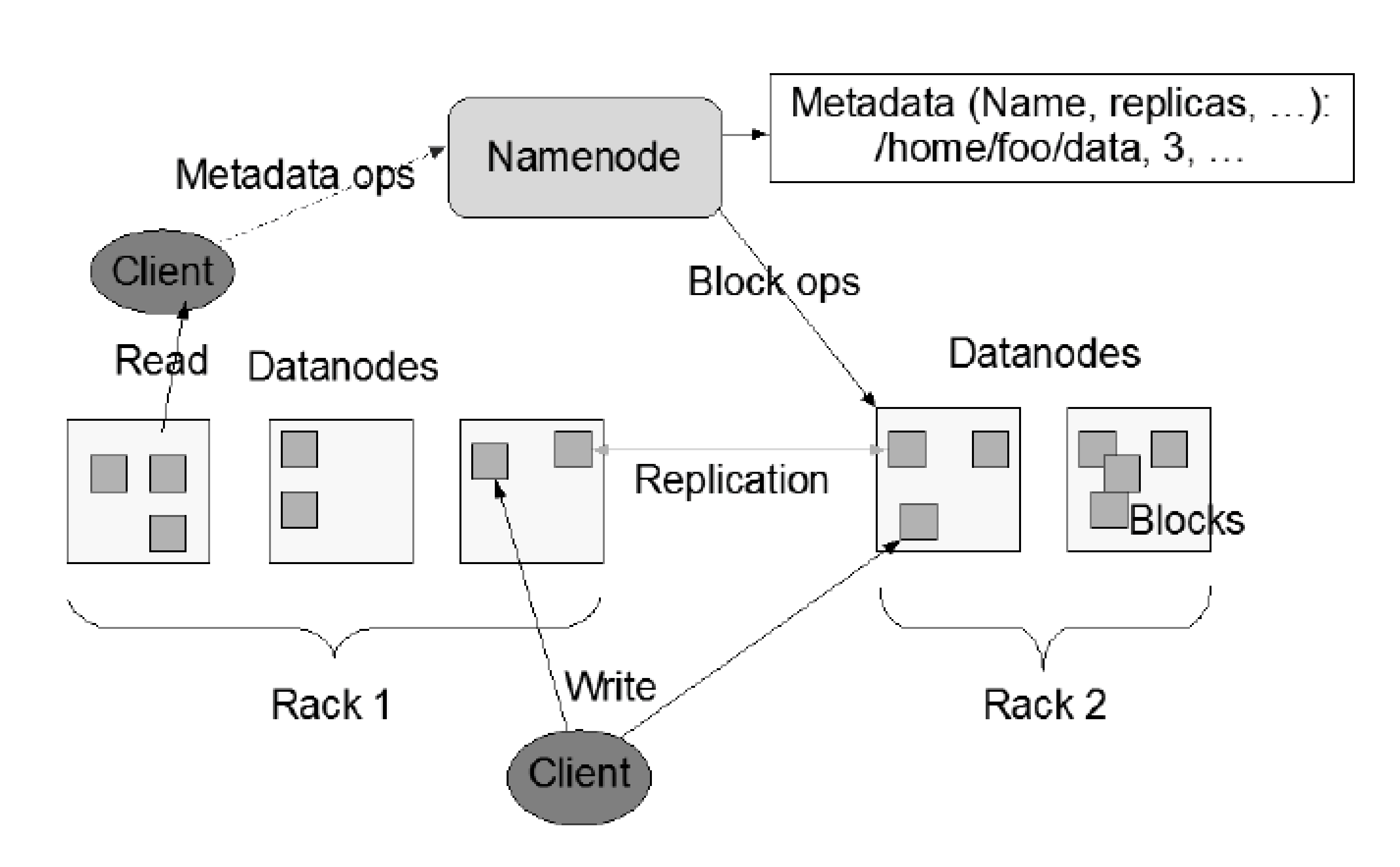


图 2-4　HDFS 的体系结构

由图 2-4 可知，名字节点 NameNode 上保存着控制数据节点 DataNode 信息的元数据(Metadata)。客户端 Client 可以通过 NameNode 对元数据进行操作，也可以直接对 DataNode 进行读和写操作。

1. NameNode 的主要功能

(1) 管理元数据信息。元数据信息包括名字空间、文件到文件块的映射、文件块到数据节点的映射三部分。管理文件块包括创建新文件块、文件复制、移除无效文件块以及回收孤立文件块等内容。

(2) 管理文件系统的命名空间。任何对文件系统元数据产生修改的操作，NameNode都会使用事务日志记录(下称 EditLog)来表示；同样地，修改文件的副本系数也将往 EditLog中插入一条记录，NameNode 将 EditLog 存储在本地操作系统的文件系统中。同时，文件系统的命名空间被存储在一个称为映像文件(FsImage)的文件中，包括文件的属性、文件块到文件的映射以及文件块到数据节点的映射等内容，FsImage 文件也是存放在 NameNode 所在的本地文件系统中。

(3) 监听请求。指监听客户端事件和 DataNode 事件。客户端事件包含名字空间的创建和删除，文件的创建、读写、重命名和删除，文件列表信息获取等信息。DataNode 事件主要包括文件块信息、心跳响应、出错信息等。处理请求指处理上面的监听请求事件并返回结果。

(4) 心跳检测。DataNode 会定期将自己的负载情况通过心跳信息向 NameNode 汇报。NameNode 全权管理数据块的复制，它周期性地从集群中的每个 DataNode 接收心跳信号和块状态报告(Block Report)。接收到心跳信号意味着该 DataNode 节点工作正常。块状态报告包含了一个该 DataNode 上所有数据块的列表。

NameNode 决定是否将文件映射到 DataNode 的复制块上。

对于最常见的三个复制块，第一个复制块存储在同一机架的不同节点上，最后一个复制块存储在不同机架的某个节点上。

实际的 I/O 事务并没有经过 NameNode，只有表示 DataNode 和块的文件映射的元数据经过 NameNode。当外部客户机发送请求，要求创建文件时，NameNode 会以块标识和该块的第一个副本的 DataNode IP 地址作为响应。这个 NameNode 还会通知其他将要接收该块的副本的 DataNode。

NameNode 在 FsImage 文件中存储所有关于文件系统名称空间的信息，包含所有事务的记录文件 EditLog 存储在 NameNode 的本地文件系统上。FsImage 和 EditLog 文件也需要复制副本，以防文件损坏或 NameNode 系统丢失。

2. DataNode 的主要功能

(1) 数据块的读写。一般是文件系统客户端需要请求对指定的 DataNode 进行读写操作，DataNode 通过 DataNode 的服务进程与文件系统客户端打交道。同时，DataNode 进程与NameNode 统一结合，对是否需要对文件块的创建、删除、复制等操作进行指挥与调度，当与 NameNode 交互过程中收到了可以执行文件块的创建、删除或复制操作的命令后，才开始让文件系统客户端执行指定的操作。具体文件的操作并不是 DataNode 来实际完成的，而是经过 DataNode 许可后，由文件系统客户端进程来执行实际操作。

(2) 向 NameNode 报告状态。每个 DataNode 节点会周期性地向 NameNode 发送心跳信号和文件块状态报告，以便 NameNode 获取到工作集群中 DataNode 节点状态的全局视图，从而掌握它们的状态。如果存在 DataNode 节点失效的情况，NameNode 会调度其他 DataNode

执行失效节点上文件块的复制处理，保证文件块的副本数达到规定数量。

(3) 执行数据的流水线复制。当文件系统客户端从 NameNode 服务器进程中获取到要进行复制的数据块列表(列表中包含指定副本的存放位置，亦即某个 DataNode 节点)后，会首先将客户端缓存的文件块复制到第一个 DataNode 节点上，此时，并非整个块都复制到第一个 DataNode 完成以后才复制到第二个 DataNode 节点上，而是由第一个 DataNode 向第二个 DataNode 节点复制，如此反复进行下去，直到完成文件块及其块副本的流水线复制。

2.2.2 HDFS 的数据操作

HDFS 被设计成在一个大集群中可以跨机器地可靠地存储海量的文件。它将每个文件存储成 block(即数据块)序列，除了最后一个 block，其他所有的 block 都是同样的大小。

1. 数据写入

在 HDFS 文件系统上创建并写一个文件的流程如图 2-5 所示。

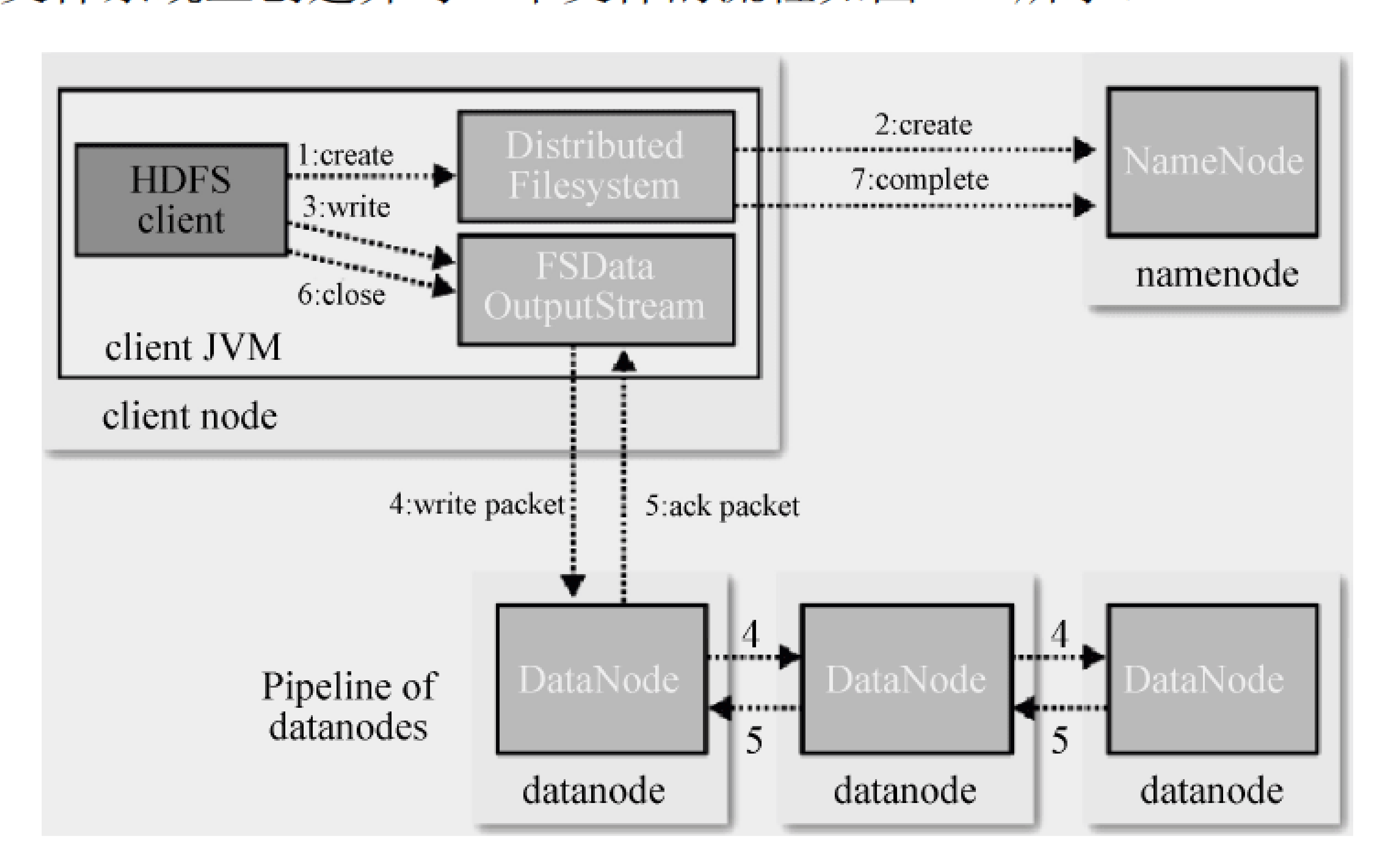


图 2-5 HDFS 写入流程

具体流程描述如下。

(1) Client 调用 DistributedFileSystem 对象的 create 方法，创建一个文件输出流(FSDataOutputStream)对象。

(2) 通过 DistributedFileSystem 对象与 Hadoop 集群的 NameNode 进行一次远程调用(RPC)，在 HDFS 的 Namespace 中创建一个文件条目(Entry)，该条目没有任何的数据块。

(3) 通过 FSDataOutputStream 对象，向 DataNode 写入数据，数据首先被写入 FSDataOutputStream 对象内部的 Buffer 中，然后数据被分割成一个个 Packet 数据包。

(4) 以 Packet 为最小单位，基于 Socket 连接发送到按特定算法选择的 HDFS 集群中的一组 DataNode(正常是 3 个，可能大于等于 1)中的一个节点上，在这组 DataNode 组成的 Pipeline 上依次传输 Packet。

(5) 这组 DataNode 组成的 Pipeline 反方向上发送 ack 确认，最终由 Pipeline 中第一个 DataNode 节点将 Pipeline ack 发送给 Client。

(6) 完成向文件写入数据，Client 在文件输出流(FSDataOutputStream)对象上调用 close

方法，关闭流。

(7) 调用 DistributedFileSystem 对象的 complete 方法，通知 NameNode 文件写入成功。

小提示

写文件过程中，Client/DataNode 与 NameNode 进行的 RPC 调用。

① 写文件开始时创建文件：Client 调用 create，在 NameNode 节点的命名空间中创建一个标识该文件的条目。

② 在 Client 连接 Pipeline 中第一个 DataNode 节点之前，Client 调用 addBlock 分配一个数据块。

③ 如果与 Pipeline 中第一个 DataNode 节点连接失败，Client 调用 abandonBlock 放弃一个已经分配的数据块。

④ 一个 Block 已经写入到 DataNode 节点磁盘，Client 调用 fsync 让 NameNode 持久化数据块的位置信息数据。

⑤ 文件写完以后，Client 调用 complete 方法通知 NameNode 写入文件成功。

⑥ DataNode 节点接收到并成功持久化一个数据块的数据后，调用 blockReceived 方法通知 NameNode 已经接收到数据块。

2. 数据读取

相比于写入流程，HDFS 文件的读取过程比较简单，如图 2-6 所示。

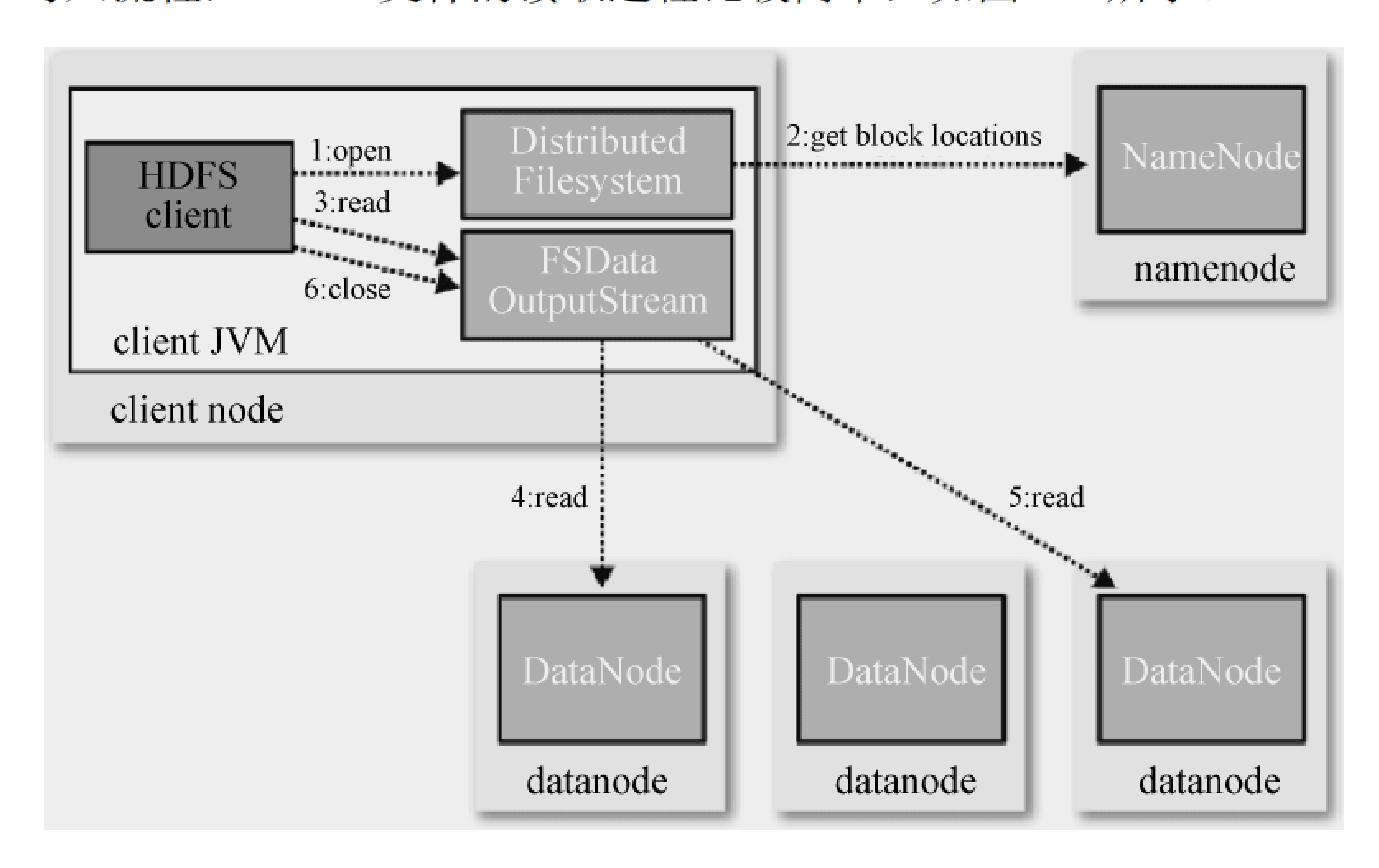


图 2-6　HDFS 读取数据

文件的读取操作流程如下。

(1) 客户端调用 FileSystem 的 open()函数打开文件，DistributedFileSystem 用 RPC 调用元数据节点，得到文件的数据块信息。

(2) 对于每一个数据块，元数据节点返回保存数据块的数据节点的地址。DistributedFileSystem 返回 FSDataInputStream 给客户端，用来读取数据。

(3) 客户端调用 stream 的 read()函数开始读取数据。

(4) DFSInputStream 连接保存此文件第一个数据块的最近的数据节点。

(5) Data 从数据节点读到客户端，当此数据块读取完毕时，DFSInputStream 关闭与此数据节点的连接，然后连接此文件下一个数据块的最近的数据节点。

(6) 当客户端读取数据完毕的时候，调用 FSDataInputStream 的 close 函数。

在读取数据的过程中，如果客户端在与数据节点通信时出现错误，则尝试连接包含此数据块的下一个数据节点。失败的数据节点将被记录下来，以后不再连接。

2.2.3 访问权限

HDFS 实现了一个与 POSIX 类似的文件和目录的权限模型。有三类权限模式：只读权限(r)、写入权限(w)和可执行权限(x)。每个文件和目录都有所属用户(owner)、所属组(group)和模式(mode)。文件或目录对其所有者、同组的其他用户以及所有其他用户分别有着不同的权限。对文件而言，当读取这个文件时，需要有 r 权限，当写入或者追加到文件时，需要有 w 权限。对目录而言，当列出目录内容时，需要具有 r 权限，当新建或删除子文件或子目录时，需要有 w 权限，当访问目录的子节点时，需要有 x 权限。不同于 POSIX 模型，HDFS 权限模型中的文件没有 sticky、setuid 或 setgid 位，因为这里没有可执行文件的概念。

每个访问 HDFS 的用户进程的标识分为两个部分，分别是用户名和组名列表。每次用户进程访问一个文件或 home 目录，HDFS 都要对其进行权限检查：

- 如果用户是 home 的所有者，则检查所有者的访问权限。
- 如果 home 关联的组在组名列表中出现，则检查组用户的访问权限；否则检查 home 其他用户的访问权限。
- 如果权限检查失败，则客户的操作会失败。

在 HDFS 中，客户端用户身份是通过宿主操作系统给出的。

对类 Unix 系统来说：

- 用户名等于‘whoami’。
- 组列表等于‘bash -c groups’。

每次文件或目录操作都传递完整的路径名给 NameNode，每一个操作都会对此路径做权限检查。客户框架会隐式地将用户身份和与 NameNode 的连接关联起来，从而减少改变现有客户端 API 的需求。经常会有这种情况：当对一个文件的某一操作成功后，之后同样的操作却会失败，这是因为文件或路径上的某些目录可能已经不复存在了。比如，客户端首先开始读一个文件，它向 NameNode 发出一个请求以获取文件第一个数据块的位置。但接下去获取其他数据块的第二个请求可能会失败。另一方面，删除一个文件并不会撤销客户端已经获得的对文件数据块的访问权限。而权限管理能使得客户端对一个文件的访问许可在两次请求之间被收回。重复一下，权限的改变并不会撤销当前客户端对文件数据块的访问许可。

如果权限检查失败，所有使用一个路径参数的方法都可能抛出 AccessControlException 异常。

2.2.4　通信协议簇

HDFS 所有的通信协议都是构建在 TCP/IP 协议上的。客户端通过一个可配置的端口连接到 NameNode，通过 ClientProtocol 与 NameNode 交互。而 DataNode 是使用 DataNodeProtocol 与 NameNode 交互的。从 ClientProtocol 和 DataNodeProtocol 抽象出一个远程调用，在设计上，NameNode 不会主动发起 RPC，而是响应来自客户端和 DataNode 的 RPC 请求。

HDFS 中的主要通信协议见表 2-1。

表 2-1　HDFS 的主要通信协议

名　称	功　能
ClientProtocol	用户进程(包括客户端进程与 DataNode 进程)与 NameNode 进程之间进行通信所使用的协议
DataNodeProtocol	DataNode 进程与 NameNode 进程进行之间通信所使用的协议，例如发送心跳报告和块状态报告
ClientDatanodeProtocol	客户端进程和 DataNode 进程之间在通信过程中所使用的协议，用户通过 ClientProtocol 协议，可操纵 HDFS 的目录命名空间、打开与关闭文件流等
NameNodeProtocol	NameNode 进程与 SecondaryNameNode 进程之间进行通信的协议
DataTransferProtocol	负责客户端与数据节点之间的数据传输
InterDatanodeProtocol	DataNode 进程之间进行通信的协议，例如客户端进程启动复制数据块，此时可能需要在 DataNode 节点之间进行块副本的流水线复制操作

(1)　ClientProtocol。

ClientProtocol 协议是用户进程(包括客户端进程与 DataNode 进程)与 NameNode 进程之间进行通信所使用的协议。当客户端进程想要与 NameNode 进程进行通信的时候，需要通过 org.apache.hadoop.hdfs.DistributedFileSystem 类，基于 ClientProtocol 协议来实现交互过程。用户代码通过 ClientProtocol 协议，可以操纵 HDFS 的目录命名空间、打开与关闭文件流等。

该接口协议中定义的与文件内容相关的操作主要有：①文件管理，文件的增、删、改，权限控制、文件块管理等；②文件系统管理，查看文件系统状态和设置元数据信息，例如容量、块大小、副本因子数等；③持久会话类，如放弃对指定块的操作、客户端同步等。

协议位置如图 2-7 所示。

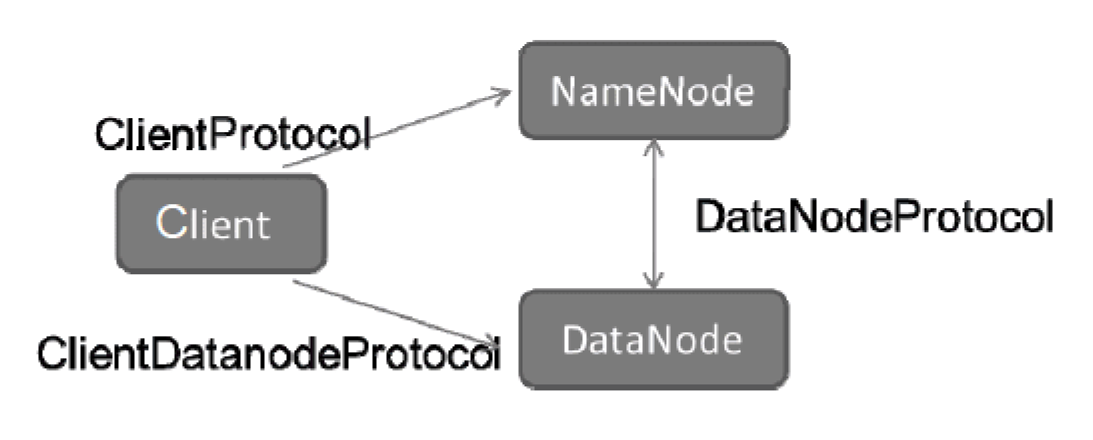


图 2-7　HDFS 协议示意

(2)　DataNodeProtocol。

该协议是用于 DataNode 与 NameNode 进行通信的协议，例如发送心跳报告和块状态报

告。一般来说，NameNode 不直接对 DataNode 进行 RPC 调用，如果一个 NameNode 需要与 DataNode 通信，唯一的方式，就是通过调用该协议接口定义的方法。

(3) ClientDatanodeProtocol。

当客户端进程需要与 DataNode 进程进行通信的时候，需要基于该协议。该协议接口定义数据块恢复的方法。

(4) NameNodeProtocol。

该协议接口定义了备用 NameNode(Secondary NameNode)与 NameNode 进行通信所需进行的操作。其中，Secondary NameNode 是一个用来辅助 NameNode 的服务器端进程，主要是对映像文件执行特定的操作，另外，还包括获取指定 DataNode 上块的操作。

(5) DataTransferProtocol。

该协议用于客户端与 DataNode 之间通信，主要实现文件块的读写及验证等操作。

(6) InterDatanodeProtocol。

该协议是 DataNode 进程之间进行通信的协议，例如客户端进程启动复制数据块，此时可能需要在 DataNode 节点之间进行块副本的流水线复制操作。

2.2.5 HDFS 的高可用性

在 Hadoop 2.0 之前的版本中，NameNode 在 HDFS 集群中存在单点故障，每一个集群中存在一个 NameNode，如果 NameNode 所在的机器出现了故障，那么，将导致整个集群无法利用，直到 NameNode 重启或者在另一台主机上启动 NameNode 守护线程。在可预知的情况下(比如 NameNode 所在的机器硬件或者软件需要升级)以及在不可预测的情况下，如果 NameNode 所在的服务器崩溃了，都将导致整个集群无法使用。

在 Hadoop 2.0 及以后的版本中，HDFS 的高可用性(High Availability)通过在同一个集群中运行两个 NameNode 实现：活动节点(Active NameNode)和备用节点(Standby NameNode)，允许在服务器崩溃或者机器维护期间快速地启用一个新的 NameNode 来恢复故障。在典型的 HA 集群中，通常有两台不同的机器充当 NameNode。在任何时间都只有一台机器处于活动(Active)状态；另一台处于待命(Standby)状态。Active NameNode 负责集群中所有客户端的操作；而 Standby NameNode 主要用于备用，它维持足够的状态，在必要时提供快速的故障恢复。

图 2-8 展示了 HDFS 的高可用性实现原理，其中，NameNode 简写为 NN，DataNode 简写为 DN。由图中可以看出，两个 NameNode 都与一组称为 JNs(JournalNodes)的互相独立的守护进程保持通信，实现 Standby NN 的状态和 Active NN 的状态同步，使元数据保持一致。当 Active NN 执行任何有关命名空间的修改时，需要发送到一半以上的 JNs 上(通过 Edits log 进行持久化存储)。当 Standby NN 观察到 Edits log 的变化时，它会从 JNs 中读取 edits 信息，并更新其内部的命名空间。一旦 Active NN 出现故障，Standby NN 首先确保自己在发生故障之前从 JNs 中读出了全部的修改内容，然后切换到 Active 状态。

为了提供快速的故障恢复，Standby NN 也需要保存集群中各个文件块的存储位置。为了达到这一目的，DataNodes 上需要同时配置这两个 NameNode 的地址，同时，与它们都建立心跳连接，并把 block 位置等信息发送给它们。对于 JNs 而言，任何时候，只允许一个 NameNode 作为数据写入者。对于 DataNodes，只执行 Active NN 发送过来的命令。

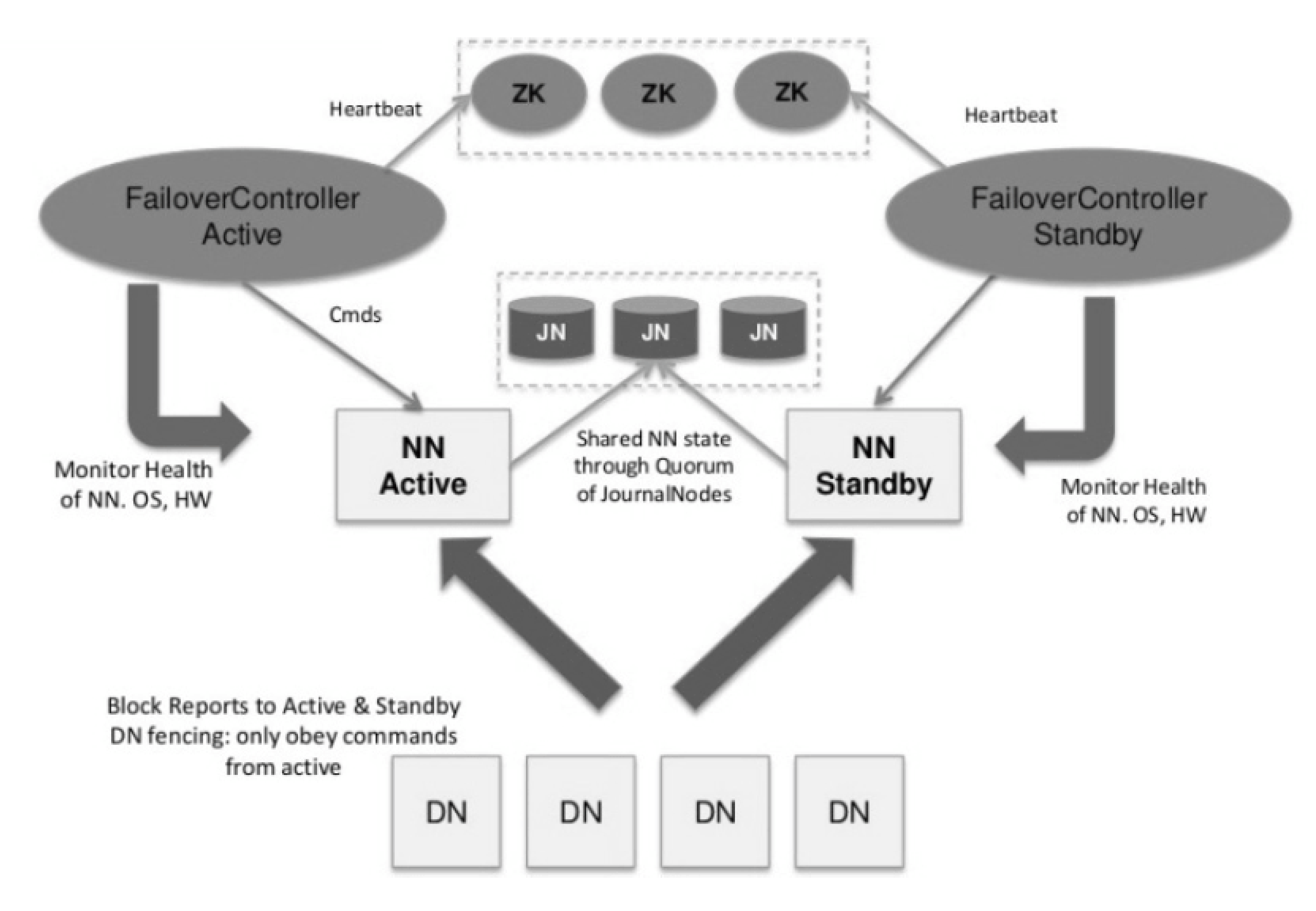


图 2-8　HDFS 高可用性的实现

2.2.6　集中缓存管理

HDFS 采用集中式的缓存管理(HDFS centralized cache management)技术。

HDFS 集中式缓存管理是一个明确的缓存机制，它允许用户指定缓存的 HDFS 路径。NameNode 会与保存着所需块数据的所有 DataNode 通信，并指导它们把块数据放在堆外缓存(off-heap)中。HDFS 集中式缓存管理的架构如图 2-9 所示。

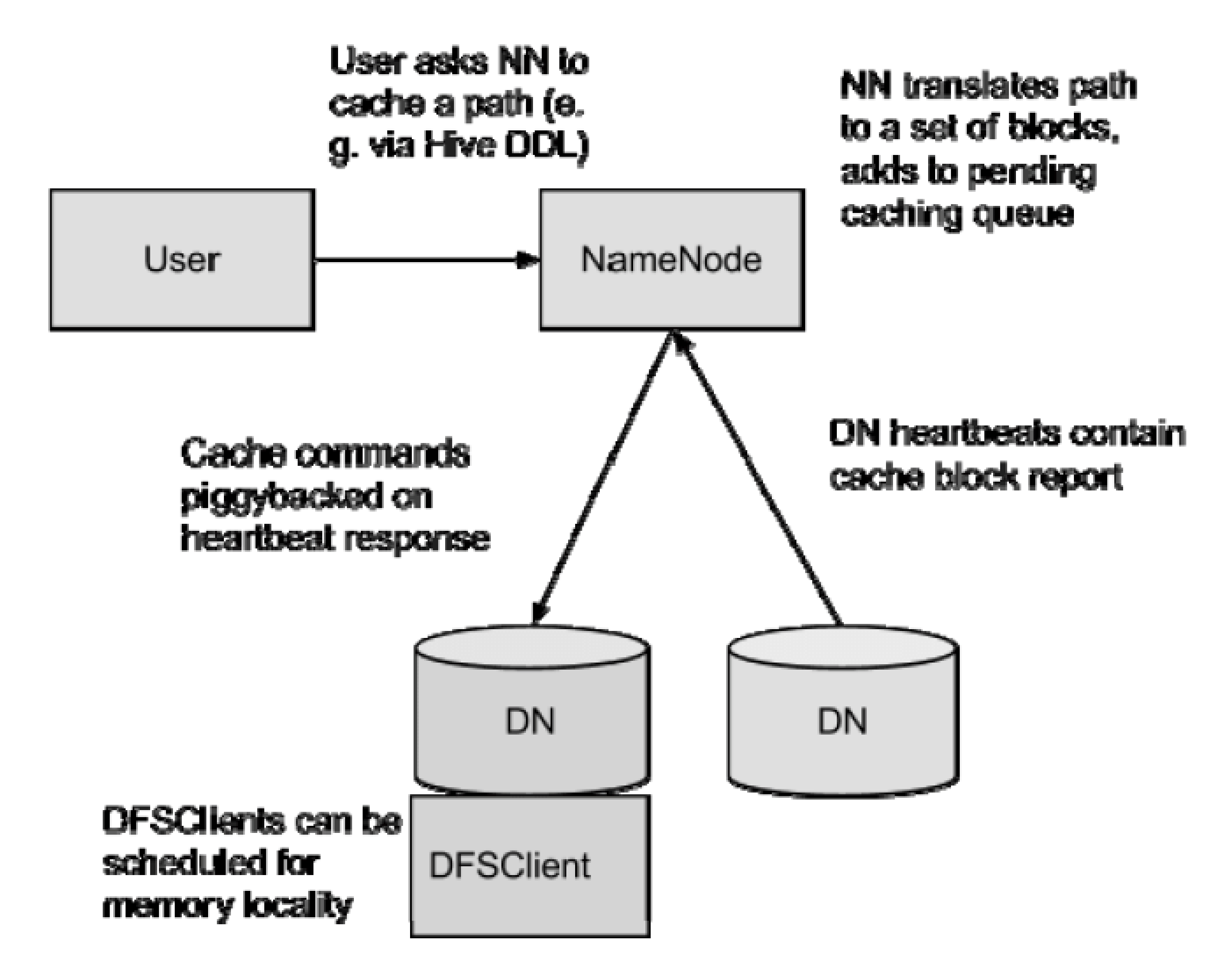


图 2-9　HDFS 集中式缓存的架构

由图 2-9 可以看到，NameNode 负责协调集群中所有 DataNode 的 off-heap 缓存。NameNode 周期性地接收来自每个 DataNode 的缓存报告，缓存报告中描述了缓存在给定 DataNode 中的所有块的信息。

NameNode 通过借助 DataNode 心跳上的缓存和非缓存命令，来管理 DataNode 缓存。缓存指令存储在 fsimage 和 editlog 中，可以通过 Java 和命令行 API 添加、移除或修改，NameNode 查询自身的缓存指令集，来确定应该缓存哪个路径。NameNode 还存储了一组缓存池(缓存池是一个管理实体，用于管理缓存指令组)。

NameNode 周期性地重复扫描命名空间和活跃的缓存，以确定需要缓存或不缓存哪个块，并向 DataNode 分配缓存任务。重复扫描也可以由用户动作来触发，比如添加或删除一条缓存指令，或者删除一个缓存池。

HDFS 集中化缓存管理具有许多优势。

(1) 用户可以根据自己的逻辑指定一些经常被使用的数据或者高优先级任务对应的数据，让它们常驻内存而不被淘汰到磁盘。当工作集的大小超过了主内存大小(这种情况对于许多 HDFS 负载都是常见的)时，这一点尤为重要。

(2) 由于 DataNode 缓存是由 NameNode 管理的，所以在分配任务时，应用程序可以通过查询一组缓存块的位置，把任务和缓存块副本放在同一位置上，提高读操作的性能。

(3) 当数据块已经被 DataNode 缓存时，客户端就可以使用一个新的更高效的零拷贝读操作 API。因为缓存数据的校验和只需由 DataNode 执行一次，所以，使用零拷贝 API 时，客户端基本上不会有开销。

(4) 集中式的缓存可以提高整个集群的内存使用率。当依赖于单独的 DataNode 上操作系统的内存进行缓存时，重复读取一个块数据会导致该块的一个或多个副本全部被送入内存中缓存。使用集中化缓存管理，用户就能明确地只锁定这 N 个副本中的 M 个了，从而节省了(N−M)个内存的使用量。

(5) 即使出现缓存数据的 DataNode 节点宕机、数据块移动或集群重启等问题，缓存都不会受到影响。因为缓存被 NameNode 统一管理并被持久化到 fsimage 和 editlog 中，出现问题后，NameNode 会调度其他存储了这个数据副本的 DataNode，把它读取到内存。

零拷贝

零拷贝(zero-copy)是实现主机或路由器等设备高速网络接口的主要技术。零拷贝技术通过减少或消除关键通信路径影响速率的操作，降低数据传输的操作系统开销和协议处理开销，从而有效提高通信性能，实现高速数据传输。

零拷贝技术可以减少数据拷贝和共享总线操作的次数，消除通信数据在存储器之间不必要的中间拷贝过程，有效地提高通信效率，是设计高速接口通道、实现高速服务器和路由器的关键技术之一。数据拷贝受制于传统的操作系统或通信协议，限制了通信性能。采用零拷贝技术，通过减少数据拷贝次数，简化协议处理的层次，在应用和网络间提供更快的数据通路，可以有效地降低通信延迟，增加网络的吞吐率。

2.2.7 日志和检查点

Hadoop 中有两个非常重要的文件 fsimage 和 edits，前面已经粗略地介绍了一些，这里做一个详细的讲解。它们位于 NameNode 的$dfs.namenode.name.dir/current/文件夹中。在 current 目录中，我们可以看到存在大量的以 edits 开头的文件和少量的以 fsimage 开头的文

件，如图 2-10 所示。

```
[root@master current]# ls |uniq -w6 -c
  30398 edits_0000000000000000189-0000000000000000190
      2 fsimage_0000000000000000000
      1 seen_txid
      1 VERSION
[root@master current]# 
```

图 2-10　current 目录

对 edits 和 fsimage 文件的概念说明如下。

(1)　edits 文件存放的是 Hadoop 文件系统的所有更新操作的日志，HDFS 文件系统客户端执行的所有写操作首先会被记录到 edits 文件中。

(2)　fsimage 文件是 Hadoop 文件系统元数据的一个永久性检查点，其中包含 Hadoop 文件系统中的所有目录和文件的索引节点序列化信息。对于文件来说，包含的信息有修改时间、访问时间、块大小信息等；对于目录来说，包含的信息主要有修改时间、访问控制权限等信息。fsimage 并不包含 DataNode 的信息，而是包含 DataNode 上块的映射信息，并存放到内存中，当一个新的 DataNode 加入到集群中时，DataNode 都会向 NameNode 提供块的信息，而 NameNode 会定期地索取块的信息，以使得 NameNode 拥有最新的块映射。

其中，edits 负责保存自最新检查点后命名空间的变化，起到日志的作用；而 fsimage 则保存了最新的元数据检查点信息。

fsimage 和 edits 文件都是经过序列化的。在 NameNode 启动的时候，会将 fsimage 文件中的内容加载到内存中，然后再执行 edits 文件中的各项操作，使得内存中的元数据与实际的同步。存在于内存中的元数据支持客户端的读操作。

NameNode 启动后，HDFS 中的更新操作会重新写到 edits 文件中。对于一个文件来说，当所有的写操作完成以后，在向客户端发送成功代码之前，将同步更新 edits 文件。在 NameNode 运行期间，由于 HDFS 的所有更新操作都是直接写到 edits 中的，时间长了会导致 edits 文件变得很大。

在 Hadoop 1.x 中，通过 SecondaryName 合并 fsimage 和 edits，以此来减小 edits 文件的大小，从而减少了 NameNode 重启的时间。在 Hadoop 2.x 中，已经不用 SecondaryName，通过配置 HA 机制实现，即在 standby NameNode 节点上运行一个叫作 CheckpointerThread 的线程，这个线程调用 StandbyCheckpointer 类的 doWork()函数，每隔一定的时间(可配置)做一次合并操作。

edits 和 fsimage 文件中的内容使用普通文本编辑器是无法直接查看的，为此，Hadoop 准备了专门的工具，用于查看文件的内容，分别为 oev 和 oiv。

oev 是 offline edits viewer(离线 edits 查看器)的缩写，该工具只操作文件，并不需要 Hadoop 集群处于运行状态。oev 提供了几个输出处理器，用于将输入文件转换为相关格式的输出文件，可以使用参数-p 指定。目前支持的输出格式有 binary(Hadoop 使用的二进制格式)、xml(在不使用参数 p 时的默认输出格式)和 stats(输出 edits 文件的统计信息)。由于没有与 stats 格式对应的输入文件，所以，一旦输出为 stats 格式，将不能再转换为原有格式。比如输入格式为 binary，输出格式为 xml，可以通过将输入文件指定为原来的输出文件，将输出文件指定为原来的输入文件，实现 binary 和 xml 的转换，而 stats 则不可以。

oev 的具体语法可以通过在命令行输入“hdfs oev”来查看，如图 2-11 所示。

```
[ root@master sbin]# hdfs oev
Usage:  bin/hdfs oev [OPTIONS] -i INPUT_FILE -o OUTPUT_FILE
Offline edits viewer
Parse a Hadoop edits log file INPUT_FILE and save results
in OUTPUT_FILE.
Required command line arguments:
-i,--inputFile <arg>    edits file to process, xml (case
                        insensitive) extension means XML format,
                        any other filename means binary format
-o,--outputFile <arg>   Name of output file. If the specified
                        file exists, it will be overwritten,
                        format of the file is determined
                        by -p option

Optional command line arguments:
-p,--processor <arg>    Select which type of processor to apply
                        against image file, currently supported
                        processors are: binary (native binary format
                        that Hadoop uses), xml (default, XML
                        format), stats (prints statistics about
                        edits file)
-h,--help               Display usage information and exit
-f,--fix-txids          Renumber the transaction IDs in the input,
                        so that there are no gaps or invalid                         transaction ID
s.
-r,--recover            When reading binary edit logs, use recovery
                        mode.  This will give you the chance to skip
                        corrupt parts of the edit log.
```

图 2-11　oev 的具体语法

oiv 是 offline image viewer 的缩写，用于将 fsimage 文件的内容转储到指定文件中，以便于阅读，该工具还提供了只读的 WebHDFS API，以允许离线分析和检查 Hadoop 集群的命名空间。oiv 在处理非常大的 fsimage 文件时是相当快的，如果不能够处理 fsimage，它会直接退出。oiv 不具备向后兼容性，比如使用 Hadoop 2.4 版本的 oiv 不能处理 hadoop 2.3 版本的 fsimage，只能使用 Hadoop 2.3 版本的 oiv。与 oev 一样，oiv 也不需要 Hadoop 集群处于运行状态。oiv 的具体语法可以通过在命令行输入“hdfs oiv”来查看。

如果 fsimage 丢失或者损坏了，我们将失去文件到块的映射关系，也就无法使用 DataNode 上的所有数据了。因此，定期及时地备份 fsimage 和 edits 文件非常重要。

fsimage 和 edit log 是 HDFS 的核心数据结构。这些文件的损坏会导致整个集群的失效。因此，NameNode 可以配置成支持多个 fsimage 和 edit log 的副本。任何 fsimage 和 edit log 的更新都会同步到每一份副本中。同步更新多个 edit log 副本会降低 NameNode 的命名空间事务处理速率。但是这种降低是可以接受的，因为 HDFS 程序中大量产生的是数据请求，而不是元数据请求。NameNode 重新启动时，会选择最新一致的 fsimage 和 edit log。

2.2.8　HDFS 快照

在 Hadoop 2.x 版本中，HDFS 提供了支持元数据快照的解决方案。

快照(Snapshot)支持存储在某个时间的数据复制，当 HDFS 数据损坏时，可以回滚到过去一个已知正确的时间点。

快照分为两种：一种是建立文件系统的索引，每次更新文件不会真正改变文件，而是新开辟一个空间用来保存更改的文件；一种是复制所有的文件系统。HDFS 把元数据和数据分离，元数据被存储在单独的 NameNode 上，实际的数据被复制并扩散到整个集群。使用

单个节点来管理元数据，使我们能够使用一个单一的逻辑时钟，建立元数据快照。

HDFS 的快照是在某一时间点对指定文件系统复制，可以是整个文件系统的，也可以是文件系统的一部分。快照采用只读模式，对重要数据进行恢复、防止用户错误性的操作。HDFS 的快照有以下特征。

(1) 快照的创建是瞬间的，代价为 O(1)，取决于子节点扫描文件目录的时间。

(2) 当且仅当快照的文件目录下有文件更新时，才会占用小部分内存，占用内存的大小为 O(M)，其中，M 为更改文件或者目录的数量。

(3) 新建快照时，DataNode 中的 block 不会被复制，快照中只是记录了文件块的列表和大小等信息。

(4) 快照不会影响正常文件系统的读写等操作。对做快照之后的数据进行的更改将会按照时间顺序逆序记录下来，用户访问的还是当前最新的数据，快照里的内容为快照创建的时间点时文件的内容减去当前文件的内容。

2.3　HDFS 的数据存储

前面主要介绍了 HDFS 系统的运行机制和原理，本节将介绍 HDFS 系统中的文件数据是如何存储和管理的。

2.3.1　数据完整性

I/O 操作过程中，难免会出现数据丢失或脏数据，数据传输的量越大，出错的概率越高。校验错误最常用的办法，就是传输前计算一个校验和，传输后计算一个校验和，两个校验和如果不相同，就说明数据存在错误。为了保证数据的完整性，一般采用下列数据校验技术：①奇偶校验技术；②MD5、SHA1 等校验技术；③CRC-32 循环冗余校验技术；④ECC 内存纠错校验技术。其中，比较常用的错误校验码是 CRC-32。

HDFS 将一个文件分割成一个或多个数据块，这些数据块被编号后，由名字节点保存，通常需要记录的信息包括文件的名称、文件被分成多少块、每块有多少个副本、每个数据块存放在哪个数据节点上、其副本存放于哪些节点上，这些信息被称为元数据。

HDFS 为了保证数据的完整性，采用校验和(checksum)检测数据是否损坏。当数据第一次引入系统时计算校验和，并且在一个不可靠的通道中传输的时候，再次检验校验和。但是，这种技术并不能修复数据(注意：校验和也可能损坏，但是，由于校验和小得多，所以可能性非常小)。数据校验和采用的是 CRC-32，任何大小的数据输入都可以通过计算，得出一个 32 位的整数校验和。

DataNode 在接收到数据后存储该数据及其校验和，或者将数据和校验和复制到其他的 DataNode 上。当客户端写数据时，会将数据及其 DataNode 发送到 DataNode 组成的管线，最后一个 DataNode 负责验证校验和，如果有损坏，则抛出 ChecksumException，这个异常属于 IOException 的子类。客户端读取数据的时候，也会检验校验和，会与 DataNode 上的校验和进行比较。每个 DataNode 上面都会有一个用于记录校验和的日志。客户端验证完之后，会告诉 DataNode，然后更新这个日志。

不仅客户端在读写数据的时候验证校验和，每个 DataNode 也会在后台运行一个 DataBlockScanner，从而定期检查存储在该 DataNode 上面的数据块。

如果客户端发现有 block 坏掉，按照以下步骤进行恢复。

(1) 客户端在抛出 ChecksumException 之前，会把坏的 block 和 block 所在的 DataNode 报告给 NameNode。

(2) NameNode 把这个 block 标记为已损坏，这样，NameNode 就不会把客户端指向这个 block，也不会复制这个 block 到其他的 DataNode。

(3) NameNode 会把一个好的 block 复制到另外一个 DataNode。

(4) NameNode 把坏的 block 删除。

HDFS 会存储每个数据块的副本，可以通过数据副本来修复损坏的数据块。客户端在读取数据块时，如果检测到错误，首先向 NameNode 报告已损坏的数据块及其正在尝试读取操作的这个 DataNode。NameNode 会将这个数据块标记为已损坏，对这个数据块的请求会被 NameNode 安排到另一个副本上。之后，它安排这个数据块的另一个副本复制到另一个 DataNode 上，如此，数据块的副本因子又回到期望水平。此后，已损坏的数据块副本会被删除。

Hadoop 的 LocalFileSystem 执行客户端的校验和验证。当写入一个名为 filename 的文件时，文件系统客户端会明确地在包含每个文件块校验和的同一个目录内建立一个名为 filename.crc 的隐藏文件。

2.3.2 数据压缩

Hadoop 作为一个较通用的海量数据处理平台，每次运算都会需要处理大量的数据。使用文件和数据压缩技术有明显的优点：①节省数据占用的磁盘空间；②加快数据在磁盘和网络中的传输速度，从而提高系统的处理速度。我们来了解一下 Hadoop 中的文件压缩。

Hadoop 支持多种压缩格式。我们可以把数据文件压缩后再存入 HDFS，以节省存储空间。在表 2-2 中，列出了几种压缩格式。

表 2-2 Hadoop 中的压缩格式

压缩格式	Unix 工具	算 法	文件扩展名	多文件支持	可 分 割
Deflate	无	Deflate	. deflate	No	No
Gzip	gzip	Deflate	. gz	No	No
Zip	zip	Deflate	. zip	Yes	Yes
Bzip	bzip2	Bzip2	. bz2	No	Yes
LZO	lzop	LZO	. lzo	No	No

所有的压缩算法都存在空间与时间的权衡：更快的压缩速率和解压速率是以牺牲压缩率为代价的。Deflate 算法是同时使用了 LZ77 与哈夫曼编码的一个无损数据压缩算法，源代码可以在 zlib 库中找到。Gzip 算法是以 Deflate 算法为基础扩展出来的一种算法。Gzip 在时间和空间上比较适中，Bzip2 算法压缩比 Gzip 更有效，但速度更慢。Bzip2 的解压速度比它的压缩速度要快，但与其他压缩格式相比，又是最慢的，但压缩效果明显是最好的。

使用压缩，有两个比较麻烦的地方：第一，有些压缩格式不能被分块、并行地处理，比如 Gzip；第二，另外的一些压缩格式虽然支持分块处理，但解压的过程非常缓慢，使作业瓶颈转移到了 CPU 上，例如 Bzip2。LZO 是一种既能够被分块并且并行处理速度也非常快的压缩算法。在 Hadoop 中，使用 LZO 压缩算法可以减小数据的大小并缩短数据的磁盘读写时间，在 HDFS 中存储压缩数据，可以使集群能保存更多的数据，延长集群的使用寿命。不仅如此，由于 MapReduce 作业通常瓶颈都在 I/O 上，存储压缩数据就意味着更少的 I/O 操作，作业运行更加高效。例如，将压缩文件直接作为入口参数交给 MapReduce 处理，MapReduce 会自动根据压缩文件的扩展名来自动选择合适的解压器处理数据。处理流程如图 2-12 所示。

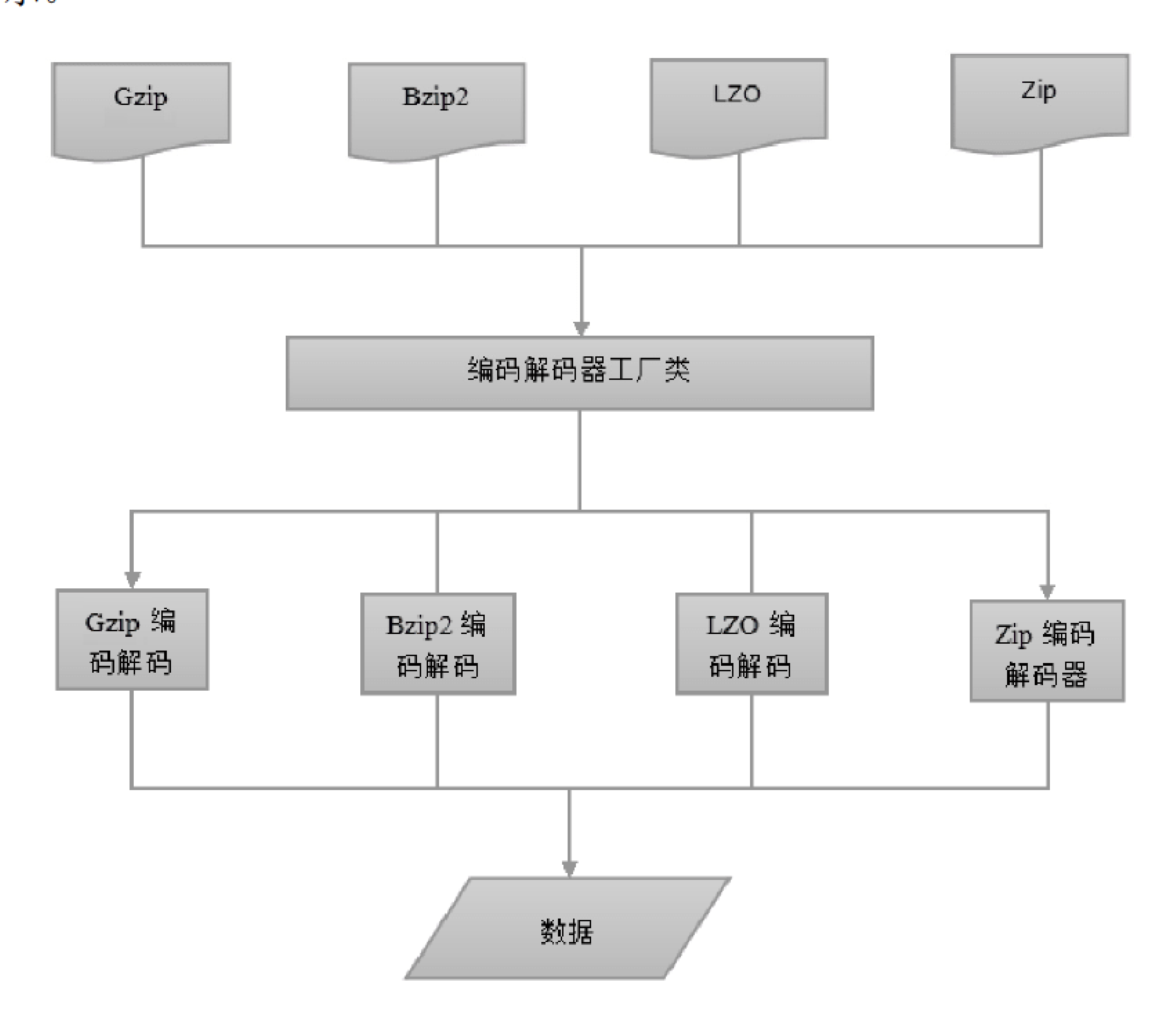


图 2-12　MapReduce 的压缩框架

LZO 的压缩文件是由许多小的 blocks 组成(约 256KB)，使得 Hadoop 的作业可以根据 block 的划分来分块工作(split job)。不仅如此，LZO 在设计时就考虑到了效率问题，它的解压速度是 Gzip 的两倍，这就让它能够节省很多的磁盘读写，它的压缩比不如 Gzip，大约压缩出来的文件比 Gzip 压缩的大一半，但是，这仍然比没有经过压缩的文件要节省 20%~50%的存储空间，这样，就可以在效率上大大地提高作业执行的速度。

在考虑如何压缩由 MapReduce 程序将要处理的数据时，压缩格式是否支持分割是很重要的。比如，存储在 HDFS 中的未压缩的文件大小为 1GB，HDFS 的块大小为 64MB，所以该文件将被存储为 16 块，将此文件用作输入的 MapReduce 作业，会创建 1 个输入分片(split，也称为“分块”。对应 block，我们统一称为“块”)，每个分片都被作为一个独立 map 任务的输入，单独进行处理。

现在假设该文件是一个 Gzip 格式的压缩文件，压缩后的大小为 1GB。与前面一样，HDFS

将此文件存储为 16 块。然而，针对每一块创建一个分块是没有用的，因为不可能从 Gzip 数据流中的任意点开始读取，map 任务也不可能独立于其他分块只读取一个分块中的数据。Gzip 格式使用 Deflate 算法来存储压缩过的数据，Deflate 将数据作为一系列压缩过的块进行存储。但是，每块的开始没有指定用户在数据流中任意点定位到下一个块的起始位置，而是其自身与数据流同步。因此，Gzip 不支持分割(块)机制。

在这种情况下，MapReduce 不分割 Gzip 格式的文件，因为它知道输入是 Gzip 压缩格式的(通过文件扩展名得知)，而 Gzip 压缩机制不支持分割机制。这样是以牺牲本地化为代价的：一个 map 任务将处理 16 个 HDFS 块。大都不是 map 的本地数据。与此同时，因为 map 任务少，所以作业分割的粒度不够细，从而导致运行时间变长。

在我们假设的例子中，如果是一个 LZO 格式的文件，我们会遇到同样的问题，因为基本压缩格式不为 reader 提供方法使其与流同步。但是，Bzip2 格式的压缩文件确实提供了块与块之间的同步标记(一个 48 位的 PI 近似值)，因此它支持分割机制。

对于文件的收集，这些问题会稍有不同。Zip 是存档格式，因此，它可以将多个文件合并为一个 Zip 文件。每个文件单独压缩，所有文档的存储位置存储在 Zip 文件的尾部。这个属性表明 Zip 文件支持文件边界处分割，每个分片中包括 Zip 压缩文件中的一个或多个文件。

2.3.3 序列化

序列化是指将结构化对象转换成字节流，以便于进行网络传输，或写入持久存储的过程。与之相对的反序列化，就是将字节流转化为一系列结构化对象的过程。

(1) 序列化有以下特征。

- 紧凑：可以充分利用稀缺的带宽资源。
- 快速：通信时大量使用序列化机制，因此，需要减少序列化和反序列化的开销。
- 可扩展：随着通信协议的升级而可升级。
- 互操作：支持不同开发语言的通信。

(2) 序列化的主要作用如下：

- 作为一种持久化格式。
- 作为一种通信的数据格式，支持不同开发语言的通信。
- 作为一种数据拷贝机制。

Hadoop 的序列化机制与 Java 的序列化机制不同，它实现了自己的序列化机制，将对象序列化到流中，值得一提的是，Java 的序列化机制是不断地创建对象，但在 Hadoop 的序列化机制中，用户可以复用对象，减少了 Java 对象的分配和回收，提高了应用效率。

在分布式系统中，进程将对象序列化为字节流，通过网络传输到另一进程，另一进程接收到字节流，通过反序列化，转回到结构化对象，以实现进程间通信。在 Hadoop 中，Mapper、Combiner、Reducer 等阶段之间的通信都需要使用序列化与反序列化技术。举例来说，Mapper 产生的中间结果<key: value1, value2...>需要写入到本地硬盘，这是序列化过程(将结构化对象转化为字节流，并写入硬盘)，而 Reducer 阶段，读取 Mapper 的中间结果的过程则是一个反序列化过程(读取硬盘上存储的字节流文件，并转回为结构化对象)。需要注意的是，能够在网络上传输的只能是字节流，Mapper 的中间结果在不同主机间洗牌时，对象将经历序列化和反序列化两个过程。

序列化是Hadoop核心的一部分，在Hadoop中，位于org.apache.hadoop.io包中的Writable接口是Hadoop序列化格式的实现，Writable接口提供两个方法：

```
public interface Writable {
    void write(DataOutput out) throws IOException;
    void readFields(DataInput in) throws IOException;
}
```

不过，没有提供比较功能，需要进行比较的话，要实现WritableComparable接口：

```
public interface WritableComparable<T> extends Writable, Comparable<T>
{ }
```

Hadoop的Writable接口是基于DataInput和DataOutput实现的序列化协议，紧凑(高效使用存储空间)、快速(读写数据、序列化与反序列化的开销小)。

Hadoop 中的键(key)和值(value)必须是实现了 Writable 接口的对象(键还必须实现WritableComparable，以便进行排序)。

Hadoop自身提供了多种具体的Writable类，包含了常见的Java基本类型(boolean、byte、short、int、float、long和double等)和集合类型(BytesWritable、ArrayWritable和MapWritable等)，如图2-13所示。

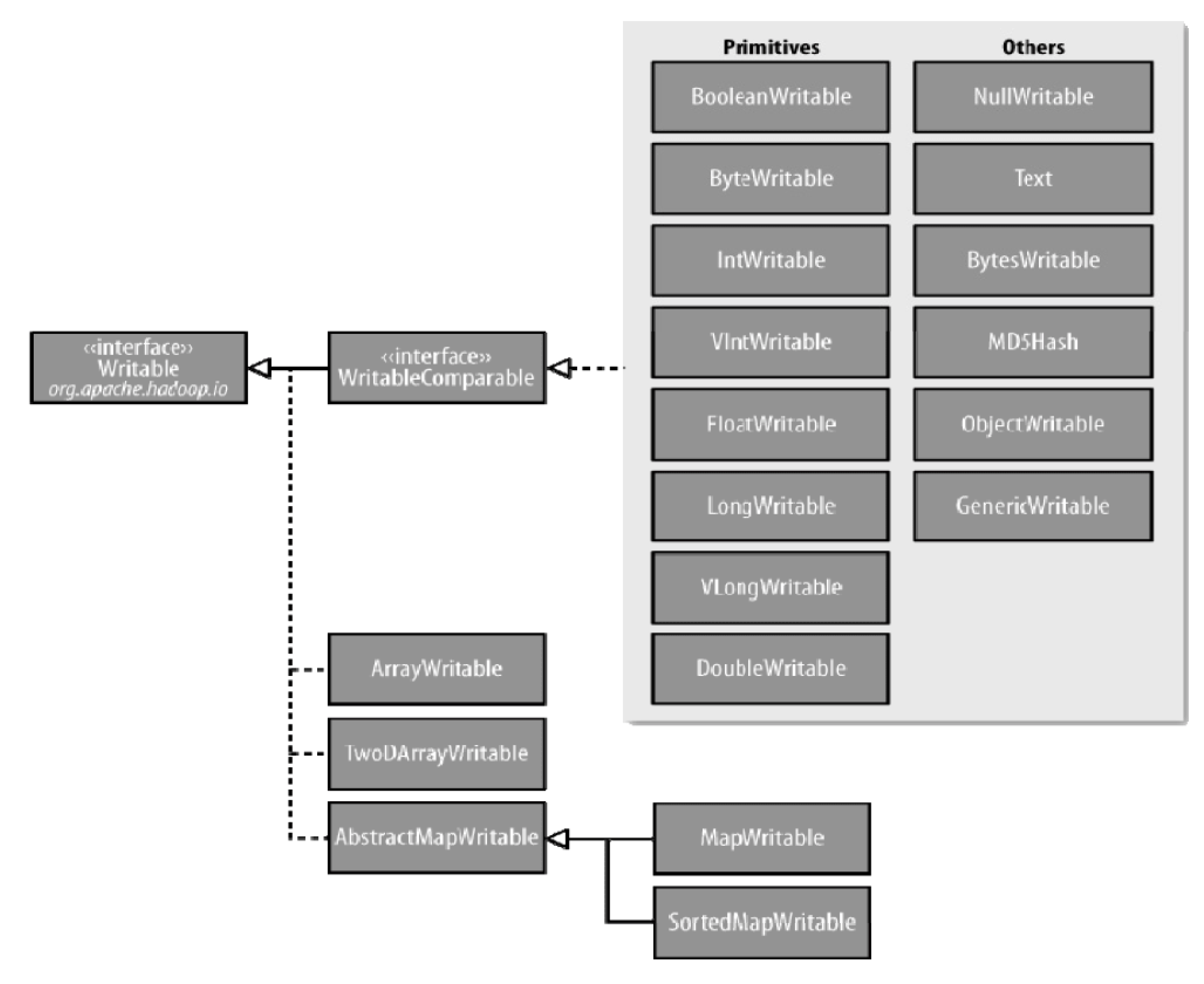


图 2-13 Writable 接口

Text： Text是UTF-8的Writable，可以理解为与java.lang.String相类似的Writable。Text类替代了UTF-8类。Text是可变的，其值可以通过调用set()方法来改变。最大可

以存储 2GB 的大小。

NullWritable：NullWritable 是一种特殊的 Writable 类型，它的序列化长度为零，可以用作占位符。

BytesWritable：BytesWritable 是一个二进制数据数组封装，序列化格式是一个 int 字段。BytesWritable 是可变的，其值可以通过调用 set()方法来改变。

ObjectWritable：ObjectWritable 适用于字段使用多种类型时。

ArrayWritable 和 TwoDArrayWritable 是针对数组和二维数组的。

MapWritable 和 SortedMapWritable 是针对 Map 和 SortMap 的。

虽然 Hadoop 内建了多种 Writable 类供用户选择，Hadoop 对 Java 基本类型的包装 Writable 类实现的 RawComparable 接口，使得这些对象不需要反序列化过程，便可以在字节流层面进行排序，从而大大缩短了比较的时间开销。但是，当我们需要更加复杂的对象时，Hadoop 的内建 Writable 类就不能满足我们的需求了(需要注意的是 Hadoop 提供的 Writable 集合类型并没有实现 RawComparable 接口，因此 也不满足我们的需要)，这时，我们就需要定制自己的 Writable 类，特别在将其作为键(key)的时候更应该如此，以求实现更高效的存储和快速的比较。

2.4　HDFS 的安装和配置

HDFS 通过一个 NameNode 作为 master 来统筹管理多个作为 slaves 的 DataNode，是 Hadoop 的核心功能组件。安装完 Hadoop 后，即完成了 HDFS 的安装。HDFS 为分布式计算存储提供了底层支持，功能及用法类似于本地文件系统。

2.4.1　Hadoop 的安装

HDFS 是 Hadoop 的核心项目，安装程序已经包含在 Hadoop 核心程序包中，从一定意义上讲，HDFS 的安装配置与 Hadoop 是一致的。

Hadoop 集群安装主要有以下工作。

1. 准备工作

(1) 首先从官网下载一个 Hadoop 程序包。一般 Hadoop 分为两个压缩文件，一个是源代码，一个是编译好的程序包。用户可根据需要选择不同的版本下载安装。

(2) 安装 Linux 服务器和必要的软件。Hadoop 既可以支持单台服务器的伪分布式部署，也可以多台集群配置部署。必需的软件主要有 Java、SSH 等。

(3) 对 Linux 进行必要的系统配置。如主机名、DNS、环境变量等。

2. 安装 Java 并配置 SSH 无密码访问

通过配置 SSH 实现基于公钥方式的无密码登录，具体操作步骤为：创建一个新的 hadoop 账户，生成这个账户的 SSH 公钥，配置公钥授权文件及设置 SSH 服务登录方式等。

3. 解压安装 Hadoop 安装包

将安装软件解压到集群内的所有机器上。

通常，安装路径要一致，一般用 HADOOP_HOME 指代安装的根路径，集群里的所有机器的 HADOOP_HOME 路径相同。

如果集群内机器的环境完全一样，可以在一台机器上配置好，然后将整个文件夹拷贝到其他机器的相同位置即可。

4. 配置 hadoop-env.sh 文件

包含 Hadoop 启动的时候配置的环境变量，包括 Hadoop 自身配置、JDK 等的设置。

5. 配置 core-site.xml、hdfs-site.xml、mapred-site.xml 等文件

配置信息将在后面详细介绍。

6. 格式化 HDFS 文件系统

这类似于 Windows 文件系统使用前需要格式化。

7. 启动所有节点并测试

本章中，我们将下载使用 Hadoop 2.6.0 版本，在 CentOS 6.5 上分别搭建伪分布式环境(即在单独的一台服务器上安装运行所有的 Hadoop 功能组件)和集群环境作为实验平台。

(1) 伪分布式部署。

① 安装 CentOS 6.5 操作系统，如图 2-14 所示。



图 2-14　安装 CentOS 操作系统

② 检查 Java 安装情况，如图 2-15 所示。如未安装，可用 rpm 包或者 yum 等方式安装 Java。这里，我们使用 Java 1.7 版本。

```
[ root@hadoop ~]# rpm -qa | grep java
java-1.6.0-openjdk-javadoc-1.6.0.0-1.66.1.13.0.el6.x86_64
eclipse-mylyn-java-3.4.2-9.el6.x86_64
java-1.6.0-openjdk-1.6.0.0-1.66.1.13.0.el6.x86_64
ant-javamail-1.7.1-13.el6.x86_64
java-1.5.0-gcj-1.5.0.0-29.1.el6.x86_64
java-1.7.0-openjdk-1.7.0.45-2.4.3.3.el6.x86_64
java-1.6.0-openjdk-devel-1.6.0.0-1.66.1.13.0.el6.x86_64
lpg-java-compat-1.1.0-4.1.el6.noarch
subversion-javahl-1.6.11-9.el6_4.x86_64
java-1.7.0-openjdk-devel-1.7.0.45-2.4.3.3.el6.x86_64
tzdata-java-2013g-1.el6.noarch
java_cup-0.10k-5.el6.x86_64
[ root@hadoop ~]#
```

图 2-15　Java 安装包

③　修改 hosts 文件，设置主机名。在 hosts 文件最后，加入一行主机名与 IP 地址，如图 2-16 所示。

```
[ root@hadoop ~]# cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6

192.168.150.128 hadoop.test
[ root@hadoop ~]# hostname
hadoop.test
[ root@hadoop ~]#
```

图 2-16　hosts 文件

④　使用 useradd 命令创建“hadoop”用户，登录并配置 SSH 无密码登录，如图 2-17 所示。

```
[ hadoop@hadoop ~]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hadoop/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/hadoop/.ssh/id_rsa.
Your public key has been saved in /home/hadoop/.ssh/id_rsa.pub.
The key fingerprint is:
74:23:44:28:ce:0a:ae:9a:71:63:ff:4b:be:57:a0:84 hadoop@hadoop.test
The key's randomart image is:
+--[ RSA 2048]----+
|        oo       |
|       . ..      |
|    o o  o o     |
|.    E...o .     |
|.. . . .S.       |
| ..   .   .      |
|o +    .   .     |
|.= o o  .        |
|+    ..=+        |
+-----------------+
```

图 2-17　配置 SSH

进入.ssh 目录，将 id_rsa.pub 复制，并且命名为“authorized_keys”，然后设置文件权限，如图 2-18 所示。

```
[hadoop@h .ssh]$ cp id_rsa.pub authorized_keys
[hadoop@h .ssh]$ chmod 600 authorized_keys
[hadoop@h .ssh]$ ls
authorized_keys  id_rsa  id_rsa.pub
[hadoop@h .ssh]$
```

图 2-18　设置文件权限

⑤　将 Hadoop 安装包解压到要安装的路径中。这里，将 Hadoop 安装在/home/hadoop/用户目录下。

⑥　配置环境变量。在/etc/profile 文件中配置 Java 及 Hadoop 相关的环境变量，并使之生效。如图 2-19 所示。

```
unset i
unset -f pathmunge

export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.45.x86_64
export CLASSPATH=.:$JAVA_HOME/jre/lib/rt.jar:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
export PATH=$PATH:JAVA_HOME/bin

export HADOOP_HOME=/home/hadoop/hadoop
export PATH=${PATH}:${HADOOP_HOME}/bin:${HADOOP_HOME}/sbin
[root@hadoop hadoop]# source /etc/profile
```

图 2-19　配置环境变量

⑦　配置 Hadoop 安装文件中的环境变量。

配置 etc/hadoop/hadoop-env.sh 文件，如图 2-20 所示。

```
# The java implementation to use.
# export JAVA_HOME=${JAVA_HOME}
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.45.x86_64
```

图 2-20　etc/hadoop/hadoop-env.sh 文件

配置 etc/hadoop/core-site.xml 文件，加入如图 2-21 所示的内容。

```
<!-- Put site-specific property overrides in this file. -->

<configuration>
  <property>
       <name>hadoop.tmp.dir</name>
       <value>/home/hadoop/hadoop/tmp</value>
   </property>
   <property>
        <name>fs.defaultFS</name>
        <value>hdfs://hadoop.test:9000</value>
        </property>
</configuration>
~
~
~
-- INSERT --
```

图 2-21　配置 etc/hadoop/core-site.xml 文件

配置 etc/hadoop/hdfs-site.xml 文件，加入如图 2-22 所示的内容。

```
<!-- Put site-specific property overrides in this file. -->

<configuration>
   <property>
      <name>dfs.replication</name>
      <value>3</value>
   </property>
   <property>
      <name>dfs.namenode.name.dir</name>
      <value>file:/home/hadoop/hadoop/dfs/name</value>
   </property>
   <property>
      <name>dfs.datanode.data.dir</name>
      <value>file:/home/hadoop/hadoop/dfs/data</value>
   </property>
   <property>
      <name>dfs.permissions</name>
      <value>false</value>
   </property>
   <property>
      <name>dfs.webhdfs.enabled</name>
      <value>true</value>
   </property>

</configuration>
-- INSERT --
```

图 2-22　配置 etc/hadoop/hdfs-site.xml 文件

配置 etc/hadoop/slaves，加入 DataNode 节点，即当前主机名，如图 2-23 所示。

```
[hadoop@hadoop hadoop]$ vim slaves
[hadoop@hadoop hadoop]$ cat slaves
hadoop.test
[hadoop@hadoop hadoop]$ █
```

图 2-23　配置 etc/hadoop/slaves

⑧　在服务器上创建目录/home/hadoop/hadoop/tmp、/home/hadoop/hadoop/dfs/name、/home/hadoop/hadoop/dfs/data，并赋予读写权限。

至此，基于单台服务器的 Hadoop 伪集群安装部署完成，启动运行即可。

(2)　集群部署。

分布式集群部署与单机 Hadoop 类似，流程上稍微复杂一些。我们以安装包括 1 个 NameNode 和 3 个 DataNode 的 Hadoop 集群为例。配置见表 2-3。

表 2-3　Hadoop 集群配置

节点名称	IP	说　明
h.master	192.168.150.129	NameNode
h.slave1	192.168.150.130	DataNode
h.slave2	192.168.150.131	DataNode
h.slave3	192.168.150.132	DataNode

①　在各个节点安装 CentOS 6.5 操作系统，安装 Java、SSH 等软件，配置允许 SSH 通过防火墙或者直接关闭各个主机的防火墙；添加名为“hadoop”的 Linux 用户；修改“/etc/sysconfig/network”中的 hostname 设置。

②　配置各个主机的 IP，并修改“/etc/hosts”文件，将每个节点的主机名写入到 hosts 文件中进行解析，如图 2-24 所示。

```
[root@h hadoop]# cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6

192.168.150.129 h.master
192.168.150.130 h.slave1
192.168.150.131 h.slave2
192.168.150.132 h.slave3
[root@h hadoop]# █
```

图 2-24　修改“/etc/hosts”文件

③　配置 SSH 无密码登录。首先在各个节点的 hadoop 用户下创建密钥，然后将 4 个节点的公钥文件 id_rsa.pub 的内容放到 authorized_keys 文件里，并设置文件权限。最后测试节点之间是否能够通过 SSH 互相连通，如图 2-25 所示。

④　将 Hadoop 安装包解压到要安装的路径中。这里，我们将 Hadoop 安装在用户目录“/home/hadoop/”下面。

⑤　配置环境变量。在所有节点的“/etc/profile”文件中配置 Java 及 Hadoop 相关的环境变量，并使之生效，如图 2-26 所示。

```
[hadoop@h ~]$ ssh h.master
The authenticity of host 'h.master (192.168.150.129)' can't be established.
RSA key fingerprint is ab:71:78:f1:e8:79:e6:e8:ba:88:b1:1d:a8:21:ce:c0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'h.master,192.168.150.129' (RSA) to the list of known hosts.
Last login: Sat Apr  9 21:13:27 2016 from h.master
[hadoop@h ~]$ exit
logout
Connection to h.master closed.
[hadoop@h ~]$ ssh h.slave1
The authenticity of host 'h.slave1 (192.168.150.130)' can't be established.
RSA key fingerprint is 1f:ae:4f:24:4a:df:19:06:27:c2:f1:1a:e0:6d:c6:c2.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'h.slave1,192.168.150.130' (RSA) to the list of known hosts.
Last login: Sat Apr  9 21:17:08 2016 from h.master
[hadoop@h ~]$ exit
logout
Connection to h.slave1 closed.
[hadoop@h ~]$ ssh h.slave2
The authenticity of host 'h.slave2 (192.168.150.131)' can't be established.
RSA key fingerprint is 46:d0:94:b4:f9:be:d8:23:48:92:be:44:36:19:57:6a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'h.slave2,192.168.150.131' (RSA) to the list of known hosts.
[hadoop@h ~]$ exit
logout
Connection to h.slave2 closed.
[hadoop@h ~]$ ssh h.slave3
The authenticity of host 'h.slave3 (192.168.150.132)' can't be established.
RSA key fingerprint is 8f:b2:60:ed:ed:33:c6:58:f1:a7:6a:9b:a3:6d:26:93.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'h.slave3,192.168.150.132' (RSA) to the list of known hosts.
[hadoop@h ~]$ exit
logout
Connection to h.slave3 closed.
[hadoop@h ~]$ 
```

图 2-25　SSH 无密码登录配置

```
unset i
unset -f pathmunge

export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.45.x86_64
export CLASSPATH=.:$JAVA_HOME/jre/lib/rt.jar:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
export PATH=$PATH:JAVA_HOME/bin

export HADOOP_HOME=/home/hadoop/hadoop
export PATH=${PATH}:${HADOOP_HOME}/bin:${HADOOP_HOME}/sbin

~
-- INSERT --
```

图 2-26　profile 文件的配置

⑥　在 NameNode 节点中，修改 Hadoop 安装目录中的配置文件。

配置 etc/hadoop/hadoop-env.sh 文件，增加 Java 接口设置，如图 2-27 所示。

```
# The java implementation to use.
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.45.x86_64
```

图 2-27　配置 etc/hadoop/hadoop-env.sh 文件

配置 etc/hadoop/core-site.xml，加入如图 2-28 所示的内容。

```
<!-- Put site-specific property overrides in this file. -->

<configuration>
  <property>
       <name>hadoop.tmp.dir</name>
       <value>/home/hadoop/hadoop/tmp</value>
   </property>
   <property>
        <name>fs.defaultFS</name>
        <value>hdfs://h.master:9000</value>
        </property>
</configuration>
~
~
~
-- INSERT --
```

图 2-28　配置 etc/hadoop/core-site.xml

配置 etc/hadoop/hdfs-site.xml，加入如图 2-29 所示的内容。

```
<!-- Put site-specific property overrides in this file. -->

<configuration>
   <property>
      <name>dfs.replication</name>
      <value>3</value>
   </property>
   <property>
      <name>dfs.namenode.name.dir</name>
      <value>file:/home/hadoop/hadoop/dfs/name</value>
   </property>
   <property>
      <name>dfs.datanode.data.dir</name>
      <value>file:/home/hadoop/hadoop/dfs/data</value>
   </property>
   <property>
      <name>dfs.permissions</name>
      <value>false</value>
   </property>
   <property>
      <name>dfs.webhdfs.enabled</name>
      <value>true</value>
   </property>

</configuration>
-- INSERT --
```

图 2-29　配置 etc/hadoop/hdfs-site.xml

配置 etc/hadoop/slaves，加入 DataNode 节点，如图 2-30 所示。

```
[hadoop@h hadoop]$ vi slaves
[hadoop@h hadoop]$ cat slaves
h.slave1
h.slave2
h.slave3
[hadoop@h hadoop]$
```

图 2-30　配置 etc/hadoop/slaves

⑦　创建 Hadoop 数据目录：/home/hadoop/hadoop/tmp、/home/hadoop/hadoop/dfs/name、/home/hadoop/hadoop/dfs/data，并赋予读写权限。

⑧　将目录“/home/hadoop/Hadoop”整体复制到其他三个节点，如图 2-31 所示。

```
[root@h hadoop]# scp -r hadoop hadoop@h.slave3:/home/hadoop/
libhdfs.so                                  100%  223KB 223.1KB/s   00:00
libhdfs.so.0.0.0                            100%  223KB 223.1KB/s   00:00
libhadoop.a                                 100% 1093KB   1.1MB/s   00:00
libhadoop.so                                100%  655KB 655.4KB/s   00:00
libhadoop.so.1.0.0                            0%    0     0.0KB/s   --:-- ETA
```

图 2-31　复制目录“/home/hadoop/Hadoop”

至此，Hadoop 集群安装完成。

主要区别在于：

- SSH 无密码登录配置。需要在每一台 Hadoop 的节点上都配置 SSH 无密码登录，使节点之间能够相互访问。
- 系统配置文件。在每台服务器上进行系统配置及网络主机名配置。
- Hadoop 配置。在 Hadoop 安装包的 etc/hadoop/slaves 中加入所有数据节点。其他配置，如文件副本数量等，可按实际情况设置。

2.4.2　HDFS 的配置

Hadoop 与 HDFS 相关的配置文件主要是 core-site.xml、hdfs-site.xml、mapred-site.xml

等三个配置文件，默认情况下，这些配置文件都是空的。

1. core-site.xml 的配置

core-site.xml 是 Hadoop 的核心配置项，是全局配置，包括 HDFS 和 MapReduce 常用的 I/O 设置等。其中，涉及 HDFS 的主要配置选项见表 2-4。

表 2-4　core-site.xml 的主要配置选项

属性名称	属 性 值	说　明
hadoop.tmp.dir	/tmp/hadoop	临时文件夹，指定后需将使用到的所有子级文件夹都要手动创建出来，否则无法正常启动服务
hadoop.http.authentication.type	simple | kerberos	验证方式，默认为简单，也可自己定义 Class，需配置所有节点
hadoop.http.authentication.token.validity	36000	验证令牌的有效时间，需配置所有节点
hadoop.http.authentication.cookie.domain	domian.tld	HTTP 验证所使用的 Cookie 的域名，IP 地址访问则该项无效，必须给所有节点都配置域名
hadoop.http.authentication.simple.anonymous.allowed	true | false	简单验证专用。默认 true，允许匿名访问
hadoop.http.authentication.kerberos.keytab	/home/xianglei/hadoop.keytab	Kerberos 验证专用，密钥文件存放位置
hadoop.security.authorization	true | false	Hadoop 服务层级验证安全验证，需配合 hadoop-policy.xml 使用，配置好以后用 dfsadmin, mradmin -refreshServiceAcl 刷新生效
hadoop.security.authentication	simple | kerberos	Hadoop 本身的权限验证，非 HTTP 访问，simple 或者 kerberos
hadoop.logfile.size	1000000000	设置日志文件大小，超过则滚动新日志
hadoop.logfile.count	20	最大日志数
fs.default.name	hdfs://master:9000	定义 NameNode 节点的 URI 和端口设定
fs.checkpoint.dir	${hadoop.tmp.dir}(默认) /dfs/namesecondary	备份元数据节点目录，一般这些目录是不同的块设备，不存在的目录会被忽略掉
fs.trash.interval	10800	HDFS 垃圾箱设置，可以恢复误删除，单位是分钟数，0 为禁用
io.bytes.per.checksum	1024	每校验码所校验的字节数，不大于 io.file.buffer.size
io.skip.checksum.errors	true | false	处理序列化文件时跳过校验码错误，不抛出异常。默认是 false

续表

属性名称	属 性 值	说 明
io.serializations	org.apache.hadoop.io.serializer.WritableSerialization	序列化的编解码器
io.seqfile.compress.blocksize	1024000	块压缩的序列化文件的最小块大小，字节
io.compression.codecs	org.apache.hadoop.io.compress.DefaultCodec, com.hadoop.compression.lzo.LzoCodec, com.hadoop.compression.lzo.LzopCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.BZip2Codec, org.apache.hadoop.io.compress.SnappyCodec	Hadoop 所使用的编解码器，Gzip、Bzip2 为自带，LZO 需安装 hadoopgpl 或者 kevinweil，逗号分隔。 snappy 需要单独安装并修改 hadoop-env.sh 配置 LD_LIBRARY_PATH=snappy 类库位置
io.compression.codec.lzo.class	com.hadoop.compression.lzo.LzoCodec	LZO 所使用的压缩编码器
io.file.buffer.size	131072	用作序列化文件处理时读写 buffer 的大小
webinterface.private.actions	true | false	Web 交互的行为设定。设为 true，则 JobTracker 和 NameNode 的 tracker 网页会出现杀任务删文件等操作连接，默认是 false
topology.script.file.name	/hadoop/bin/RackAware.py	机架感知位置脚本
topology.script.number.args	1000	机架感知脚本管理的主机数，IP 地址

分布式集群部署模式时，核心配置文件 core-site.xml 的参考配置格式如下：

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://master:9000</value>
    </property>
    <property>
        <name>io.file.buffer.size</name>
        <value>131072</value>
    </property>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>file:/home/hadoop/tmp</value>
    </property>
    <property>
```

```
        <name>hadoop.proxyuser.hduser.hosts</name>
        <value>*</value>
    </property>
    <property>
        <name>hadoop.proxyuser.hduser.groups</name>
        <value>*</value>
    </property>
</configuration>
```

2. hdfs-site.xml 配置

hdfs-site.xml 是 HDFS 的局部配置文件，主要配置选项见表 2-5。

表 2-5　hdfs-site.xml 的主要配置选项

属性名称	属 性 值	说　明
dfs.namenode.handler.count	10	NameNode 启动后开启的线程数
dfs.datanode.http.address	0.0.0.0:50075	DataNode 的 tracker 页面监听地址和端口
dfs.datanode.ipc.address	0.0.0.0:50020	DataNode 的 IPC 监听端口，为 0 的话会在随机端口监听，通过心跳通知 NameNode
dfs.datanode.address	0.0.0.0:50010	DataNode 服务端口，为 0 则随机监听
dfs.datanode.max.xcievers	256	Linux 下的打开文件最大数量，当出现 DataXceiver 报错的时候，需要调大
dfs.datanode.data.dir.perm	755	DataNode 所使用的本地文件夹的路径权限，默认 755
dfs.datanode.failed.volumes.tolerated	0	DataNode 磁盘卷出错次数。0 次则任何卷出错都要停止 DataNode
dfs.name.dir	${hadoop.tmp.dir}/dfs/name	存储在本地的 NN 数据镜像的目录，作为 NameNode 的冗余备份
dfs.data.dir	/var/data/hadoop/hdfs/data1, /var/data/hadoop/hdfs/data2, /var/data/hadoop/hdfs/data3	真正的 DataNode 数据保存路径，可以多分区，逗号分隔
dfs.heartbeat.interval	3	DataNode 心跳检测时间间隔，单位为秒
dfs.client.block.write.retries	3	数据块写入最多重试次数，在此次数之前不会捕获失败
dfs.max.objects	0	dfs 最大并发对象数，HDFS 中的目录块都会被认为是一个对象，0 表示不限制
dfs.safemode.extension	30	指定系统退出安全模式时需要的延迟时间，单位为秒，默认为 30
dfs.http.address	0.0.0.0:50070	NameNode 的 Web 管理端口
dfs.https.enable	true | false(默认)	NameNode 的 tracker 是否支持 HTTPS 协议
dfs.https.address	0.0.0.0:50470	NameNode 的 tracker 页面监听访问地址
dfs.permissions	true | false	dfs 权限是否打开，一般设置为 false
dfs.permissions.supergroup	supergroup(默认)	设置 hdfs 超级权限组
dfs.replication	3	hdfs 数据块的副本数，默认 3，理论上数量越大速度越快，但需要的存储空间也更多

续表

属性名称	属 性 值	说 明
dfs.replication.max	512	DataNode 数据块的最大副本数量。DataNode 故障临时恢复时会导致数据超过默认副本数，此属性通常不用配置
dfs.replication.min	1	DataNode 数据块的最小副本数量
dfs.replication.interval	3	NameNode 计算复制 DataNode 节点的工作周期数，单位为秒。一般情况下不用指定
dfs.hosts	/usr/local/hadoop/conf /dfs.hosts.allow	DataNode 的白名单。仅在 dfs.hosts 文件中指定的 DataNode 有权限连接到 NameNode 上。如果该参数不指定，那么默认所有的 DataNode 都可以连接到 NameNode
dfs.hosts.exclude	/usr/local/hadoop/conf /dfs.hosts.deny	DataNode 黑名单

3. mapred-site.xml 配置

mapred-site.xml 用于配置 MapReduce 使用的框架，与 HDFS 的关联主要是文件和 I/O 操作。这里简单介绍一下 mapred-site.xml 的相关配置项，见表 2-6。

表 2-6 mapred-site.xml 的相关配置项

属性名称	属 性 值	说 明
mapred.local.dir	/data1/hdfs/mapred/local, /data2/hdfs/mapred/local, ...	mapred 是做本地计算所使用的文件夹，可以配置多块硬盘，以逗号分隔
mapred.system.dir	/data1/hdfs/mapred/system, /data2/hdfs/mapred/system, ...	mapred 存放控制文件所使用的文件夹，可配置多块硬盘，逗号分隔
mapred.temp.dir	/data1/hdfs/mapred/temp, /data2/hdfs/mapred/temp, ...	mapred 是共享的临时文件夹路径，可配置多块硬盘，以逗号分隔
mapred.local.dir.minspacestart	1073741824	本地运算文件夹剩余空间低于该值则不在本地做计算。单位是字节数，默认为 0
mapred.local.dir.minspacekill	1073741824	本地计算文件夹剩余空间低于该值则不再申请新任务，单位是字节数，默认为 0
hadoop.job.history.location	—	job 历史文件保存路径，无可配置参数，也不用写在配置文件中，默认在 logs 的 history 文件夹下
hadoop.job.history.user.location		用户历史文件的存放位置
io.sort.factor	30	处理流合并时的文件排序数
io.sort.mb	600	排序所使用的内存数量，单位是兆，默认为 1

2.4.3　启动 HDFS

配置完成后，需要先在 NameNode 中格式化 HDFS 系统，然后再启动。

1. HDFS 的格式化

如图 2-32 所示，我们使用“hdfs namenode -format”命令格式化 Hadoop 2.6.0 的 HDFS 系统。

```
[root@h hadoop]# hdfs namenode -format
16/04/09 22:34:26 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = h.master/192.168.150.129
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 2.6.0
STARTUP_MSG:   classpath = /home/hadoop/hadoop/etc/hadoop:/home/hadoop/hadoop/share/hadoop/com
```

……

```
16/04/09 22:40:29 INFO common.Storage: Storage directory /home/hadoop/hadoop/dfs/name has been
successfully formatted.
16/04/09 22:40:29 INFO namenode.NNStorageRetentionManager: Going to retain 1 images with txid >
= 0
16/04/09 22:40:29 INFO util.ExitUtil: Exiting with status 0
16/04/09 22:40:29 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at h.master/192.168.150.129
************************************************************/
[hadoop@h hadoop]$ 
```

图 2-32　HDFS 的格式化

2. HDFS 的启动

通过执行脚本 start-dfs.sh 或 start-all.sh 启动 Hadoop 文件系统，如图 2-33 所示。

```
[hadoop@h hadoop]$ sbin/start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [h.master]
h.master: starting namenode, logging to /home/hadoop/hadoop/logs/hadoop-hadoop-namenode-h.master.
out
h.slave2: starting datanode, logging to /home/hadoop/hadoop/logs/hadoop-hadoop-datanode-h.slave2.
out
h.slave1: starting datanode, logging to /home/hadoop/hadoop/logs/hadoop-hadoop-datanode-h.slave1.
out
h.slave3: starting datanode, logging to /home/hadoop/hadoop/logs/hadoop-hadoop-datanode-h.slave3.
out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /home/hadoop/hadoop/logs/hadoop-hadoop-secondaryn
amenode-h.master.out
starting yarn daemons
starting resourcemanager, logging to /home/hadoop/hadoop/logs/yarn-hadoop-resourcemanager-h.maste
r.out
h.slave2: starting nodemanager, logging to /home/hadoop/hadoop/logs/yarn-hadoop-nodemanager-h.sla
ve2.out
h.slave1: starting nodemanager, logging to /home/hadoop/hadoop/logs/yarn-hadoop-nodemanager-h.sla
ve1.out
h.slave3: starting nodemanager, logging to /home/hadoop/hadoop/logs/yarn-hadoop-nodemanager-h.sla
ve3.out
[hadoop@h hadoop]$ 
```

图 2-33　启动 Hadoop

3. 验证

可以使用 Java 自带的小工具 jps 查看进程，如图 2-34 所示；也可以用“hdfs dfsadmin -report”命令查看 Hadoop 集群的状态，如图 2-35 所示。

```
[hadoop@h hadoop]$ jps
2564 SecondaryNameNode
2374 NameNode
2700 ResourceManager
2907 Jps
[hadoop@h hadoop]$
```

图 2-34　jps 命令

```
[hadoop@h hadoop]$ hdfs dfsadmin -report
Configured Capacity: 44104237056 (41.08 GB)
Present Capacity: 26043969536 (24.26 GB)
DFS Remaining: 26043895808 (24.26 GB)
DFS Used: 73728 (72 KB)
DFS Used%: 0.00%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
Live datanodes (3):

Name: 192.168.150.130:50010 (h.slave1)
Hostname: h.slave1
Decommission Status : Normal
Configured Capacity: 14701412352 (13.69 GB)
DFS Used: 24576 (24 KB)
Non DFS Used: 5938585600 (5.53 GB)
DFS Remaining: 8762802176 (8.16 GB)
DFS Used%: 0.00%
DFS Remaining%: 59.61%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Sat Apr 09 23:02:15 CST 2016


Name: 192.168.150.131:50010 (h.slave2)
Hostname: h.slave2
Decommission Status : Normal
Configured Capacity: 14701412352 (13.69 GB)
DFS Used: 24576 (24 KB)
```

图 2-35　查看集群状态

打开浏览器，输入“http://localhost:50070”，页面显示正常内容，说明 Hadoop 安装成功并在运行中，如图 2-36 所示。

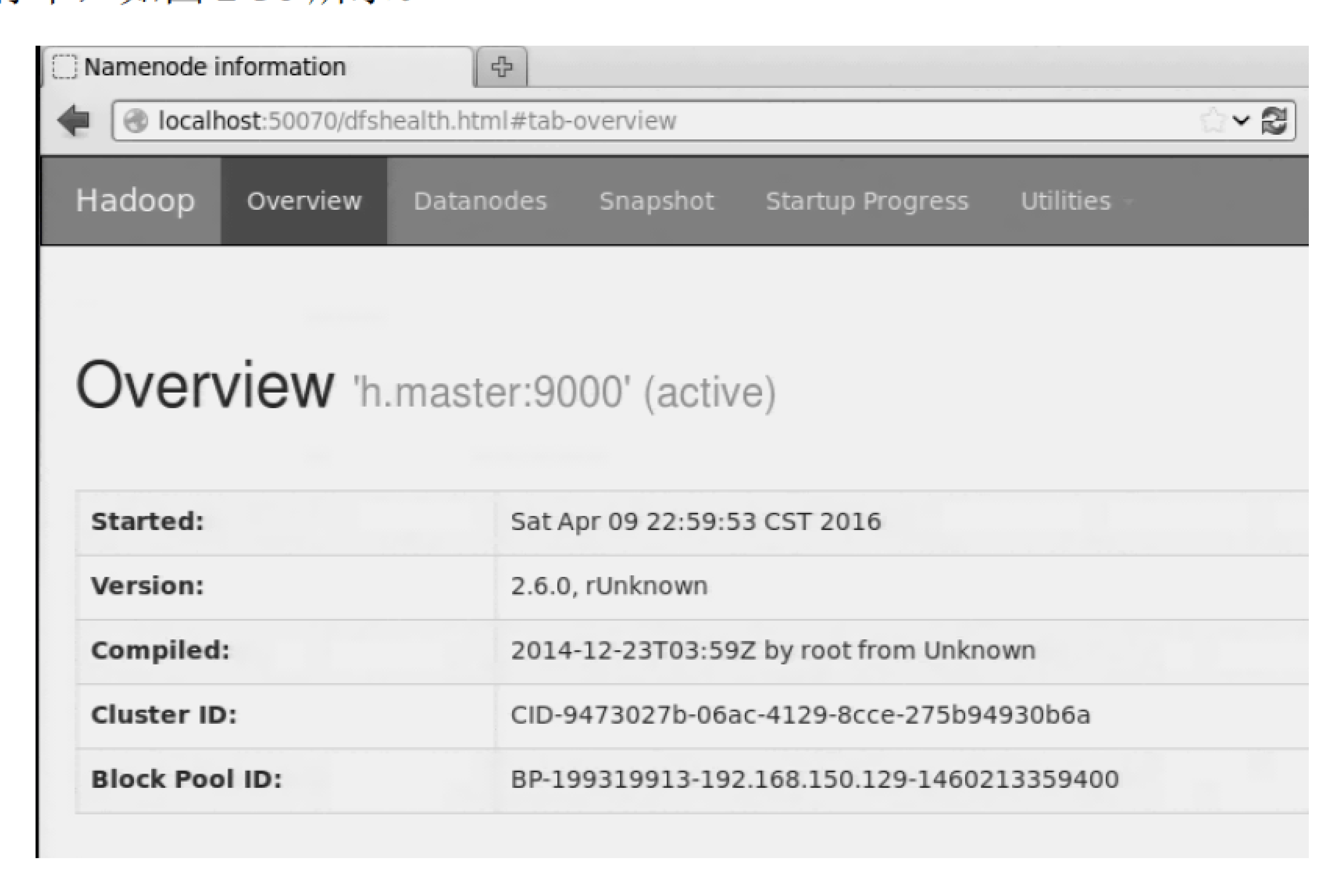


图 2-36　Hadoop Web 的状态

2.5 小　　结

本章介绍了分布式文件系统及 HDFS，并重点介绍了 HDFS 的结构组成、运行原理，以及数据操作、数据完整性、压缩存储、序列化等诸多优点和特性。

通过学习本章的内容，读者应能够对 HDFS 分布式文件系统有一定的认识，掌握 HDFS 的内部运行机制，为下一章的操作打下基础。

当然，HDFS 也有其自身的缺陷和不足，比如不适合存储大量的小文件、不适合大量的随机读文件操作等，有兴趣的读者不妨扩展一下。

本章最后详细介绍了 Hadoop 的安装配置，读者要熟练掌握并实际完成 Hadoop 集群配置，后续学习实践也将在本章安装的环境的基础上进行。

第 3 章

HDFS 操作实践

学习目标：

本章将介绍如何使用和管理 Hadoop 的 HDFS 文件系统，包括 Shell 命令接口、Java 编程接口等文件或文件夹的读写操作，实践性较强。

通过本章的学习，将使读者能够在使用中掌握 HDFS 的文件操作，为后续的学习打下基础。

本章要点：

- Shell 命令接口
- Java 接口编程开发
- WebHDFS 接口
- 文件读写操作

3.1 HDFS 接口与编程

HDFS 提供了多种用户操作和编程接口，既通过 Shell 命令管理文件与目录、管理作业调度、控制与优化集群性能等，也提供了 Java、C 语言等的编程接口，用户可以通过编写程序对 HDFS 进行扩展。

3.1.1 Shell 命令

HDFS 资源 URI 的格式如下：

```
scheme://authority/path
```

其中 scheme 是协议名，一般是 file 或 hdfs；authority 是授权访问的主机名或 IP；path 是访问路径。例如：

```
hdfs://localhost:9000/user/chunk/test.txt
```

如果已经在 core-site.xml 里配置了 fs.default.name=hdfs://localhost:9000，则仅使用 /user/chunk/test.txt 即可。

在 HDFS 的所有接口中，Shell 命令行接口最简单，也是开发者比较熟悉的方式。我们通过使用“hdfs -help”命令，可以看到 HDFS 支持的文件系统命令，如图 3-1 所示。

```
[root@test hadoop]# hdfs -help
Usage: hdfs [--config confdir] COMMAND
       where COMMAND is one of:
  dfs                  run a filesystem command on the file systems supported in Hadoop.
  namenode -format     format the DFS filesystem
  secondarynamenode    run the DFS secondary namenode
  namenode             run the DFS namenode
  journalnode          run the DFS journalnode
  zkfc                 run the ZK Failover Controller daemon
  datanode             run a DFS datanode
  dfsadmin             run a DFS admin client
  haadmin              run a DFS HA admin client
  fsck                 run a DFS filesystem checking utility
  balancer             run a cluster balancing utility
  jmxget               get JMX exported values from NameNode or DataNode.
  mover                run a utility to move block replicas across
                       storage types
  oiv                  apply the offline fsimage viewer to an fsimage
  oiv_legacy           apply the offline fsimage viewer to an legacy fsimage
  oev                  apply the offline edits viewer to an edits file
  fetchdt              fetch a delegation token from the NameNode
  getconf              get config values from configuration
  groups               get the groups which users belong to
  snapshotDiff         diff two snapshots of a directory or diff the
                       current directory contents with a snapshot
  lsSnapshottableDir   list all snapshottable dirs owned by the current user
                                                Use -help to see options
  portmap              run a portmap service
  nfs3                 run an NFS version 3 gateway
  cacheadmin           configure the HDFS cache
  crypto               configure HDFS encryption zones
  storagepolicies      get all the existing block storage policies
  version              print the version

Most commands print help when invoked w/o parameters.
```

图 3-1 HDFS 支持的文件系统命令

HDFS 支持的文件系统命令主要有两类。

(1) 用户命令：用于管理 HDFS 日常操作，如 dfs、fsck、fetchdt 等。

(2) 系统管理命令：主要用于控制和管理 HDFS 集群，如 balancer、namenode、datanode、dfsadmin、secondarynamenode 等。限于篇幅，这里只介绍几种常用的命令模块。

1. hdfs dfs [GENERIC_OPTIONS] [COMMAND_OPTIONS]

“hdfs dfs”提供了类似于 Linux Shell 一样的命令集，其用法与 Linux Shell 基本一致。下面详细介绍各个命令。

(1) appendToFile。

说明：将一个或者多个本地文件追加到目的文件。成功返回 0，错误返回 1。

格式：hdfs dfs -appendToFile <localsrc> ... <dst>

示例：

```
hdfs dfs -appendToFile localfile /user/hadoop/hadoopfile
hdfs dfs -appendToFile localfile1 localfile2 /user/hadoop/hadoopfile
hdfs dfs -appendToFile localfile hdfs://nn.example.com/hadoop/hadoopfile
```

(2) cat。

说明：将路径指定文件的内容输出到 stdout。成功返回 0，错误返回-1。

格式：hdfs dfs -cat URI [URI ...]

示例：

```
hdfs dfs -cat hdfs://nn1.example.com/file1 hdfs://nn2.example.com/file2
hdfs dfs -cat file:///file3 /user/hadoop/file4
```

(3) chgrp。

说明：改变文件所属的用户组。如果使用-R 选项，则这一操作对整个目录结构递归执行。使用这一命令的用户必须是文件的所属用户，或者是超级用户。

格式：hdfs dfs -chgrp [-R] GROUP URI [URI ...]

(4) chmod。

说明：改变文件的权限。使用-R 将使改变在目录结构下递归进行。命令的使用者必须是文件的所有者或超级用户。

格式：hdfs dfs -chmod [-R] <MODE[, MODE]... | OCTALMODE> URI [URI ...]

(5) chown。

说明：改变文件的所属用户。如果使用-R 选项，则这一操作对整个目录结构递归执行。使用这一命令的用户必须是文件在命令变更之前的所属用户，或者是超级用户。

格式：hdfs dfs -chown [-R] [OWNER][:[GROUP]] URI [URI]

(6) copyFromLocal。

说明：从本地复制，与 put 命令相似，但限定源路径是本地的。

格式：hdfs dfs -copyFromLocal <localsrc> URI

(7) copyToLocal。

说明：复制到本地，与 get 命令相似，但限定目的路径是本地的。

格式：hdfs dfs -copyToLocal [-ignorecrc] [-crc] URI <localdst>

(8) count。

说明：计算文件、目录的数量。成功返回 0，错误返回-1。

格式：hdfs dfs -count [-q] [-h] <paths>

示例：

```
hdfs dfs -count hdfs://nn1.example.com/file1 hdfs://nn2.example.com/file2
hdfs dfs -count -q hdfs://nn1.example.com/file1
hdfs dfs -count -q -h hdfs://nn1.example.com/file1
```

(9) cp。

说明：将文件从源路径复制到目标路径。这个命令允许有多个源路径，但同时，目标路径必须是一个目录。成功返回 0，错误返回-1。

格式：hdfs dfs -cp [-f] [-p | -p[topax]] URI [URI ...] <dest>

示例：

```
hdfs dfs -cp /user/hadoop/file1 /user/hadoop/file2
hdfs dfs -cp /user/hadoop/file1 /user/hadoop/file2 /user/hadoop/dir
```

(10) du。

说明：显示目录中所有文件的大小，或者当只指定一个文件时，显示此文件的大小。成功返回 0，错误返回-1。

格式：hdfs dfs -du [-s] [-h] URI [URI ...]

示例：

```
hdfs dfs -du /user/hadoop/dir1 /user/hadoop/file1
  hdfs://nn.example.com/user/hadoop/dir1
```

(11) dus。

说明：显示文件的大小。此命令可以用“du -s”替代。

格式：hdfs dfs -dus <args>

(12) expunge。

作用：清空回收站。

格式：hdfs dfs -expunge

(13) get。

说明：复制文件到本地文件系统。可用“-ignorecrc”选项复制 CRC 校验失败的文件。使用“-crc”选项复制文件以及 CRC 信息。成功返回 0，错误返回-1。

格式：hdfs dfs -get [-ignorecrc] [-crc] <src> <localdst>

示例：

```
hdfs dfs -get /user/hadoop/file localfile
hdfs dfs -get hdfs://nn.example.com/user/hadoop/file localfile
```

(14) getfacl。

说明：显示文件或者目录的权限控制列表。成功返回 0，错误返回非零值。

格式：hdfs dfs -getfacl [-R] <path>

示例：

```
hdfs dfs -getfacl /file
hdfs dfs -getfacl -R /dir
```

(15) getfattr。

说明：显示文件或者目录的扩展属性。成功返回 0，错误返回非零值。

格式：hdfs dfs -getfattr [-R] -n name | -d [-e en] <path>

示例：

```
hdfs dfs -getfattr -d /file
hdfs dfs -getfattr -R -n user.myAttr /dir
```

(16) getmerge。

说明：接受一个源目录和一个目标文件作为输入，并且将源目录中所有的文件连接成

本地目标文件。addnl 是可选的，用于指定在每个文件结尾添加一个换行符。

格式：hdfs dfs -getmerge <src> <localdst> [addnl]

(17) ls。

说明：与 Linux 中一样，返回子目录或子文件列表。成功返回 0，错误返回-1。

格式：hdfs dfs -ls [-R] <args>

示例：

```
hdfs dfs -ls /user/hadoop/file1
```

(18) lsr。

说明：ls 命令的递归版本，一般使用“ls -R”代替。

格式：hdfs dfs -lsr <args>

(19) mkdir。

说明：创建目录，加-p 选项创建多层目录。成功返回 0，错误返回-1。

格式：hdfs dfs -mkdir [-p] <paths>

示例：

```
hdfs dfs -mkdir /user/hadoop/dir1 /user/hadoop/dir2
hdfs dfs -mkdir hdfs://nn1.example.com/user/hadoop/dir
  hdfs://nn2.example.com/user/hadoop/dir
```

(20) moveFromLocal。

说明：类似 put，区别在于 put 操作完成后删除。

格式：hdfs dfs -moveFromLocal <localsrc> <dst>

(21) mv。

说明：将文件从源路径移动到目标路径。这个命令允许有多个源路径，此时，目标路径必须是一个目录。不允许在不同的文件系统间移动文件。成功返回 0，错误返回-1。

格式：hdfs dfs -mv URI [URI ...] <dest>

示例：

```
hdfs dfs -mv /user/hadoop/file1 /user/hadoop/file2
hdfs dfs -mv hdfs://nn.example.com/file1 hdfs://nn.example.com/file2
  hdfs://nn.example.com/file3 hdfs://nn.example.com/dir1
```

(22) put。

说明：从本地文件系统中复制单个或多个源路径到目标文件系统。也支持从标准输入设备中读取输入，写入目标文件系统。成功返回 0，错误返回-1。

格式：hdfs dfs -put <localsrc> ... <dst>

示例：

```
hdfs dfs -put localfile /user/hadoop/hadoopfile
hdfs dfs -put localfile1 localfile2 /user/hadoop/hadoopdir
hdfs dfs -put localfile hdfs://nn.example.com/hadoop/hadoopfile
```

(23) rm。

说明：删除指定的文件或目录。成功返回 0，错误返回-1。

格式：hdfs dfs -rm [-f] [-r|-R] [-skipTrash] URI [URI ...]

示例：

```
hdfs dfs -rm hdfs://nn.example.com/file /user/hadoop/emptydir
```

(24) rmr。

说明：rm 的递归版本，已过时，一般使用“rm -r”代替。

格式：hdfs dfs -rmr [-skipTrash] URI [URI ...]

(25) setfacl。

说明：设置文件或者目录的权限控制列表。成功返回 0，错误返回非零值。

格式：hdfs dfs -setfacl [-R] [-b|-k -m|-x <acl_spec> <path>]|[--set <acl_spec> <path>]

示例：

```
hdfs dfs -setfacl -m user:hadoop:rw- /file
hdfs dfs -setfacl -x user:hadoop /file
hdfs dfs -setfacl -b /file
hdfs dfs -setfacl -k /dir
hdfs dfs -setfacl --set user::rw-,user:hadoop:rw-,group::r--,other::r--
  /file
hdfs dfs -setfacl -R -m user:hadoop:r-x /dir
hdfs dfs -setfacl -m default:user:hadoop:r-x /dir
```

(26) setfattr。

说明：设置文件或者目录的扩展属性。成功返回 0，错误返回非零值。

格式：hdfs dfs -setfattr -n name [-v value] | -x name <path>

示例：

```
hdfs dfs -setfattr -n user.myAttr -v myValue /file
hdfs dfs -setfattr -n user.noValue /file
hdfs dfs -setfattr -x user.myAttr /file
```

(27) setrep。

说明：改变文件和目录的复制因子。成功返回 0，错误返回-1。

格式：hdfs dfs -setrep [-R] [-w] <numReplicas> <path>

示例：

```
hdfs dfs -setrep -w 3 /user/hadoop/dir1
```

(28) stat。

说明：返回指定路径的统计信息。成功返回 0，错误返回-1。

格式：hdfs dfs -stat URI [URI ...]

示例：

```
hdfs dfs -stat path
```

(29) tail。

说明：将文件尾部 1KB 的内容输出到 stdout。成功返回 0，错误返回-1。

格式：hdfs dfs -tail [-f] URI

示例：

```
hdfs dfs -tail pathname
```

(30) test

说明：检查文件。选项“-e”检查文件是否存在，如果存在则返回 0；选项“-z”检查文

件是否为 0 字节，如果是则返回 0；选项“-d”检查路径是否为目录，如果是则返回 1，否则返回 0。

格式：hdfs dfs -test -[ezd] URI

示例：

```
hdfs dfs -test -e filename
```

(31) text。

说明：将源文件输出为文本格式。允许的格式是 zip 和 TextRecordInputStream。

格式：hdfs dfs -text <src>

(32) touchz。

说明：创建一个空文件。成功返回 0，错误返回-1。

格式：hdfs dfs -touchz URI [URI ...]

示例：

```
hdfs dfs -touchz pathname
```

小提示

“hadoop dfs”与“hdfs dfs”都是操作 HDFS 文件系统的命令，“hadoop dfs”属于早期版本的格式，已经过时，一般使用“hdfs dfs”。

“hadoop fs”也是文件系统操作命令，但使用范围更广，能够操作其他格式文件系统，如 local、HDFS 等，可以在本地与 Hadoop 分布式文件系统的交互操作中使用。

2. hdfs fsck [GENERIC_OPTIONS] <path> [-list-corruptfileblocks | [-move | -delete | -openforwrite] [-files [-blocks [-locations | -racks]]]] [-includeSnapshots]

fsck 是一个文件系统健康状况检查工具，用来检查各类问题，比如，文件块丢失等(如图 3-2 所示)。但是，注意它不会主动恢复备份缺失的 block，这个是由 NameNode 单独的线程异步处理的。

```
[root@test hadoop]# hdfs fsck /
Connecting to namenode via http://test.hadoop:50070
FSCK started by root (auth:SIMPLE) from /192.168.221.133 for path / at Mon Mar 07 23:41:56 CST 20
16
Status: HEALTHY
 Total size:    0 B
 Total dirs:    1
 Total files:   0
 Total symlinks:                0
 Total blocks (validated):      0
 Minimally replicated blocks:   0
 Over-replicated blocks:        0
 Under-replicated blocks:       0
 Mis-replicated blocks:         0
 Default replication factor:    1
 Average block replication:     0.0
 Corrupt blocks:                0
 Missing replicas:              0
 Number of data-nodes:          1
 Number of racks:               1
FSCK ended at Mon Mar 07 23:41:56 CST 2016 in 2 milliseconds


The filesystem under path '/' is HEALTHY
[root@test hadoop]# 
```

图 3-2　fsck 命令的运行结果

fsck 命令的参数说明见表 3-1。

表 3-1 fsck 参数的说明

参 数	说 明
path	被检查的目录
-move	将破损的文件移至/lost+found 目录
-delete	删除破损的文件
-files	打印正在检查的文件名
-openforwrite	打印正在打开写操作的文件
-includeSnapshots	如果检测的目录有快照表，则包括快照数据
-list-corruptfileblocks	打印出丢失的块及它们所属的文件列表
-blocks	打印 block 报告(需要与-files 参数一起使用)
-locations	打印每个 block 的位置信息(需要与-files 参数一起使用)
-racks	打印数据节点位置信息的网络拓扑图(与-files 参数一起使用)

3. hdfs datanode [-regular | -rollback | -rollingupgrade rollback]

运行一个 HDFS 集群的数据节点。参数说明见表 3-2。

表 3-2 hdfs datanode 命令参数的说明

参 数	说 明
-regular	正常启动数据节点
-rollback	将数据节点回滚到前一个版本
-rollingupgrade rollback	回滚一个升级操作

4. hdfs namenode [GENERIC_OPTIONS]

“hdfs namenode”是运行 NameNode 的命令，是一个比较核心的工具。该命令的主要参数说明见表 3-3。

表 3-3 hdfs namenode 命令参数的说明

参 数	说 明
-backup	启动备份节点
-checkpoint	启动检查点
-format [-clusterid cid] [-force] [-nonInteractive]	格式化 NameNode
-upgrade [-clusterid cid] [-renameReserved <k-v pairs>]	分发新版本的 Hadoop 后，NameNode 以 upgrade 选项启动
-rollback	将 NameNode 回滚到前一版本。这个选项要在停止集群并分发老的 Hadoop 版本后使用
-finalize	保持最近的升级并删除文件系统的前一状态

续表

参　数	说　明
-importCheckpoint	从检查点目录装载镜像并保存到当前检查点目录，检查点目录由 fs.checkpoint.dir 指定
-recover [-force]	恢复损坏的文件系统上丢失的元数据

5. hdfs dfsadmin [GENERIC_OPTIONS]

dfsadmin 是一个多任务的工具，我们可以使用它来获取 HDFS 的状态信息，以及在 HDFS 上执行的一系列管理操作。该命令的主要参数说明见表 3-4。

表 3-4　hdfs dfsadmin 命令参数的说明

参　数	说　明
-report [-live] [-dead] [-decommissioning]	报告文件系统的基本信息和统计信息
-safemode enter|leave|get|wait	安全模式维护命令。安全模式是 NameNode 的一个状态，这种状态下，NameNode： 1. 不接受对命名空间的更改(只读)。 2. 不复制或删除块。 NameNode 会在启动时自动进入安全模式，当配置的块最小百分比数满足最小的副本数条件时，会自动离开安全模式。安全模式可以手动进入，但是，这样的话也必须手动关闭安全模式
-saveNamespace	保存当前命名空间到存储目录并重置编辑日志。需要在安全模式中运行
-restoreFailedStorage true|false|check	这个选项“打开/关闭”自动尝试修复故障存储副本的功能。check 选项将返回当前设置
-refreshNodes	重新读取 hosts 和 exclude 文件，使新的节点或需要退出集群的节点能够被 NameNode 重新识别。这个命令在新增节点或注销节点时用到
-setQuota <quota> <dirname>...<dirname>	为每个目录<dirname>设定配额<quota>。目录配额是一个长整型整数，强制限定了目录树下的名字个数。 命令会在这个目录上工作良好，以下情况会报错：N 不是一个正整数；用户不是管理员；这个目录不存在或是文件；目录会马上超出新设定的配额
-clrQuota <dirname>...<dirname>	为每一个目录<dirname>清除配额设定。 命令会在这个目录上工作良好，以下情况会报错：这个目录不存在或是文件；用户不是管理员。如果目录原来没有配额，则不会报错
-setStoragePolicy <path> <policyName>	设置文件或目录的存储策略
-getStoragePolicy <path>	获取文件或目录的存储策略

续表

参　数	说　明
-finalizeUpgrade	终止 HDFS 的升级操作。DataNode 删除前一个版本的工作目录，之后 NameNode 也这样做。这个操作终结整个升级过程
-metasave filename	保存 NameNode 的主要数据结构到 hadoop.log.dir 属性指定的目录下的<filename>文件。对于下面的每一项，<filename>中都会有一行内容与之对应：NameNode 收到的 DataNode 的心跳信号；等待被复制的块；正在被复制的块；等待被删除的块
-refreshServiceAcl	刷新服务级别的授权策略文件
-refreshCallQueue	刷新来自配置中的调用(call)队列
-refresh <host:ipc_port> <key> [arg1..argn]	触发一个在运行时刷新的由<host:ipc_port>上的<key>指定的资源，剩余参数随后被发送给主机
-printTopology	根据 NameNode 报告打印机架和节点树
-refreshNamenodes datanodehost:port	对于给定的数据节点，重新加载配置文件，停止服务已删除的数据块池，开始新的数据块池服务
-setBalancerBandwidth <bandwidth in bytes per second>	更改数据节点所使用的负载均衡网络带宽，该值将覆盖 dfs.balance.bandwidthPerSec 的参数值
-fetchImage <local directory>	从 NameNode 中下载最新的 fsimage，保存到指定的本地路径
-shutdownDatanode <datanode_host:ipc_port> [upgrade]	提交关机请求给指定的 DataNode
-getDatanodeInfo <datanode_host:ipc_port>	获得指定 DataNode 的信息

dfsadmin 命令的使用示例如图 3-3 所示。

```
[root@master ~]# hdfs dfsadmin -printTopology
Rack: /default-rack
   192.168.221.130:50010 (slave1.hadoop)
   192.168.221.131:50010 (slave2.hadoop)
   192.168.221.132:50010 (slave3.hadoop)
```

图 3-3　dfsadmin 命令的使用示例

6. hdfs cacheadmin

管理员和用户通过“hdfs cacheadmin”命令管理缓存资源。

缓存指令由一个唯一的无重复的 64 位整数 ID 来标识。即使缓存指令后来被删除了，ID 也不会重复使用。缓存池由一个唯一的字符串名称来标识。

(1)　增加缓存：addDirective。

用法：hdfs cacheadmin -addDirective -path <path> -pool <pool-name> [-force] [-replication <replication>] [-ttl <time-to-live>]

参数说明见表 3-5。

表 3-5　addDirective 的参数说明

参　数	说　明
-path	要缓存的路径。该路径可以是文件夹或文件
-pool	要加入缓存指令的缓存池。我们必须对该缓存池有写权限，以便添加新的缓存指令
-force	强制执行，不检查缓存池的资源限制
-replication	要使用的缓存副本因子，默认为 1
-ttl	缓存指令可以保持有效多长时间。可以按照分钟,小时,天来指定，如 30m,4h,2d。有效单位为[smhd]。never 表示永不过期的指令。如果未指定该值，那么，缓存指令就不会过期

(2)　删除一个缓存：removeDirective。

用法：hdfs cacheadmin -removeDirective <id>

参数 id 指定要删除的缓存指令的 ID。删除时，必须对该指令的缓存池拥有写权限。

(3)　删除指定路径下的每一个缓存：removeDirectives。

用法：hdfs cacheadmin -removeDirectives <path>

参数 path 中设置要删除的缓存指令的路径。删除时必须对该指令的缓存池拥有写权限。

(4)　缓存列表：listDirectives。

用法：hdfs cacheadmin -listDirectives [-stats] [-path <path>] [-pool <pool>]

参数说明见表 3-6。

表 3-6　listDirectives 的参数说明

参　数	说　明
-path	只列出带有该路径的缓存指令。注意，如果路径 path 在缓存池中有一条或多条没有读权限的缓存指令，那么它就不会被列出来
-pool	只列出该缓存池内的缓存指令
-stats	列出基于 path 的缓存指令统计信息

(5)　新增缓存池：addPool。

用法：hdfs cacheadmin -addPool <name> [-owner <owner>] [-group <group>] [-mode <mode>] [-limit <limit>] [-maxTtl <maxTtl>]

参数说明见表 3-7。

表 3-7　addPool 的参数说明

参　数	说　明
name	新缓存池的名称
-owner	该缓存池所有者的名称。默认为当前用户
-group	缓存池所属的组。默认为当前用户的主要组名
-mode	以 Unix 风格表示的该缓存池的权限。权限以八进制数表示，如 0755。默认值为 0755

续表

参　数	说　明
-limit	在该缓存池内由指令总计缓存的最大字节数。默认不设限制
-maxTtl	添加到该缓存池的指令的最大生存时间。该值以秒,分,时,天的格式来表示，如 120s,30m,4h,2d。有效单位为[smhd]。默认不设最大值。never 表示没有限制

(6) 修改缓存池：modifyPool。

用法：hdfs cacheadmin -modifyPool <name> [-owner <owner>] [-group <group>] [-mode <mode>] [-limit <limit>] [-maxTtl <maxTtl>]

参数说明见表 3-8。

表 3-8　modifyPool 的参数说明

参　数	说　明
name	要修改的缓存池的名称
-owner	该缓存池所有者的名称
-group	缓存池所属的组
-mode	以 Unix 风格表示的该缓存池的权限。八进制数形式
-limit	在该缓存池内要缓存的最大字节数
-maxTtl	添加到该缓存池的指令的最大生存时间

(7) 删除缓存池：removePool。

用法：hdfs cacheadmin -removePool <name>

参数 name 指定要删除的缓存池的名称。

(8) 缓存池列表：listPools。

用法：hdfs cacheadmin -listPools [-stats] [<name>]

参数说明见表 3-9。

表 3-9　addPool 的参数说明

参　数	说　明
name	若指定，则仅列出该缓存池的信息
-stats	显示额外的缓存池统计信息

7. hdfs balancer [-threshold <threshold>] [-policy <policy>]

HDFS 集群非常容易出现机器与机器之间磁盘利用率不平衡的情况，尤其是增加新的数据节点时。保证 HDFS 中的数据平衡非常重要。HDFS 出现不平衡的状况将引发很多问题，比如 MapReduce 程序无法很好地利用本地计算的优势、机器之间无法达到更好的网络带宽使用率等。

在 Hadoop 中，包含一个 Balancer 程序，可以调节 HDFS 集群平衡的状态。启动 Balancer 服务时，界面如图 3-4 所示。

```
[root@master sbin]# jps
3488 ResourceManager
3353 SecondaryNameNode
4812 Bootstrap
17428 Jps
[root@master sbin]# sh start-balancer.sh
starting balancer, logging to /root/hadoop/hadoop/logs/hadoop-root-balancer-master.hadoop.out
[root@master sbin]# jps
3488 ResourceManager
3353 SecondaryNameNode
4812 Bootstrap
17500 Jps
17465 Balancer
```

图 3-4　启动 Balancer 服务

服务启动后，集群管理人员可用 balancer 命令进行分析和再平衡数据，如图 3-5 所示。

```
[root@master sbin]# hdfs balancer --help
Usage: java Balancer
        [-policy <policy>]      the balancing policy: datanode or blockpool
        [-threshold <threshold>]        Percentage of disk capacity
        [-exclude [-f <hosts-file> | comma-sperated list of hosts]]     Excludes the specified da
tanodes.
        [-include [-f <hosts-file> | comma-sperated list of hosts]]     Includes only the specifi
ed datanodes.

Generic options supported are
-conf <configuration file>     specify an application configuration file
-D <property=value>            use value for given property
-fs <local|namenode:port>      specify a namenode
-jt <local|resourcemanager:port>    specify a ResourceManager
-files <comma separated list of files>    specify comma separated files to be copied to the map r
educe cluster
-libjars <comma separated list of jars>    specify comma separated jar files to include in the cl
asspath.
-archives <comma separated list of archives>    specify comma separated archives to be unarchived
 on the compute machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]
```

图 3-5　可用 balancer 命令进行分析和再平衡数据

参数 threshold 是判断集群是否平衡的目标参数，表示 HDFS 达到平衡状态的磁盘使用率偏差值。默认设置为 10，参数取值范围是 0~100。如果机器之间磁盘使用率偏差小于 10%，我们就认为 HDFS 集群已经达到了平衡的状态。

8. hdfs version

hdfs version 命令用于查看当前系统的版本，运行示例如图 3-6 所示。

```
[root@test ~]# hdfs version
Hadoop 2.6.0
Subversion Unknown -r Unknown
Compiled by root on 2014-12-23T03:59Z
Compiled with protoc 2.5.0
From source with checksum 18e43357c8f927c0695f1e9522859d6a
This command was run using /root/hadoop/hadoop/share/hadoop/common/hadoop-common-2.6.0.jar
[root@test ~]#
```

图 3-6　使用 hdfs version 命令查看当前系统的版本

3.1.2 Java 接口操作

由于 Hadoop 本身就是使用 Java 语言编写的，理论上，通过 Java API 能够调用所有的 Hadoop 文件系统的操作接口。

Hadoop 有一个抽象的文件系统概念，在 Java 抽象类 org.apache.hadoop.fs 中定义了接口。只要某个文件系统实现了这个接口，那么，它就可以作为 Hadoop 支持的文件系统。目前 Hadoop 能够支持的文件系统如表 3-10 所示。

表 3-10 Hadoop 文件类的实现

文件系统	URI 方案	Java 实现 (org.apache.hadoop)	说 明
Local	file	fs.LocalFileSystem	支持有客户端校验和的本地文件系统。没有校验和的本地文件系统在 fs.RawLocalFileSystem 中实现
HDFS	hdfs	hdfs.DistributionFileSystem	Hadoop 的分布式文件系统。将 HDFS 设计成与 MapReduce 结合使用以实现高性能
HFTP	hftp	hdfs.HftpFileSystem	支持通过 HTTP 方式以只读的方式访问 HDFS
HSFTP	hsftp	hdfs.HsftpFileSystem	支持通过 HTTPS 方式以只读的方式访问 HDFS
HAR	har	fs.HarFileSystem	构建在 Hadoop 文件系统之上，对文件进行归档。Hadoop 归档文件主要用来减少 NameNode 的内存使用
KFS	kfs	fs.kfs.KosmosFileSystem	Cloudstore 文件系统是类似于 HDFS 和 Google 的 GFS 文件系统，用 C++编写
FTP	ftp	fs.ftp.FtpFileSystem	由 FTP 服务器支持的文件系统
S3(本地)	s3n	fs.s3native.NativeS3FileSystem	基于 Amazon S3 的文件系统
S3(基于块)	s3	fs.s3.NativeS3FileSystem	基于 Amazon S3 的文件系统，以块格式存储解决了 S3 的 5GB 文件大小的限制

在 Hadoop 中，主要是定义了一组分布式文件系统和通用的 I/O 组件和接口，Hadoop 的文件系统准确地应该称作 Hadoop I/O。而 HDFS 是实现该文件接口的 Hadoop 自带的分布式文件项目，是对 Hadoop I/O 接口的实现。在处理大数据集时，为实现最优性能，通常使用 HDFS 存储。

org.apache.hadoop.fs 包由接口类(FsConstants、Syncable 等)、Java 类(AbstractFileSystem、BlockLocation、FileSystem、FileUtil、FSDataInputStream 等)、枚举类型(如 CreateFlag)、异常类(ChecksumException、InvalidPathException 等)和错误类(如 FSError)组成。每个子对象中都定义了相应的方法，通过对 org.apache.hadoop.fs 包的封装与调用，可以拓展 HDFS 应

用，更好地帮助用户使用集群海量存储。

在介绍 Java 接口操作之前，先介绍几个常用的 Java 类。

(1)　FileSystem。

org.apache.hadoop.fs.FileSystem，通用文件系统基类，用于与 HDFS 文件系统交互，编写的 HDFS 程序都需要重写 FileSystem 类。通过 FileSystem，可以非常方便地像操作本地文件系统一样操作 HDFS 集群文件。

FileSystem 提供了 get 方法，一个是通过配置文件获取与 HDFS 的连接；一个是通过 URL 指定配置文件，获取与 HDFS 的连接，URL 的格式为 hdfs://namenode/xxx.xml。

方法的原型如下：

```
public static FileSystem get(Configuration conf) throws IOException;
public static FileSystem get(URI uri, Configuration conf) throws IOException;
public static FileSystem get(final URI uri, final Configuration conf, final
  String user) throws IOException, InterruptedException;
```

其中，Configuration(org.apache.hadoop.conf.Configuration)类对象封装了客户端或服务器的配置；URI 是指文件在 HDFS 里存放的路径。

(2)　FSDataInputStream。

org.apache.hadoop.fs.FSDataInputStream，文件输入流，用于读取 HDFS 文件，它是 Java 中 DataInputStream 的派生类，支持从任意位置读取流式数据。

常用的读取方法是从指定的位置，读取指定大小的数据至缓存区。方法如下所示：

```
int read(long position, byte[] buffer, int offset, int length)
```

还有用于随时定位的方法，可以定位到指定的读取点，如下所示：

```
void seek(long desired)
```

通过 long getPos()方法，还可以获取当前的读取点。

(3)　FSDataOutputStream。

org.apache.hadoop.fs.FSDataOutputStream，文件输出流，是 DataOutputStream 的派生类，通过这个类，能够向 HDFS 顺序写入数据流。

通常的写入方法为 write，如下所示：

```
public void write(int b)
```

获取当前写入点的函数为 long getPos()。

(4)　Path。

org.apache.hadoop.fs.Path，文件与目录定位类，用于定义 HDFS 集群中指定的目录与文件绝对或相对路径。

可以通过多种方式构造 Path，如通过 URL 的模式，通常编写方式为：

```
hdfs://ip:port/directory/filename
```

Path 可以与 FileSystem 的 open 函数相关联，通过 Path 构造访问路径，用 FileSystem 进行访问。

(5)　FileStatus。

org.apache.hadoop.fs.FileStatus，文件状态显示类，可以获取文件与目录的元数据、长度、

块大小、所属用户、编辑时间等信息，同时，可以设置文件用户、权限等内容。

FileStatus 有很多 get 与 set 方法，如获取文件长度的 long getLen()方法、设置文件权限的 setPermission(FsPermission permission)方法等。

下面，我们开始 Hadoop 的 Java 操作之旅。

1. 创建文件

FileSystem 类里提供了很多 API，用来创建文件，其中，最简单的一个是：

```
public FSDataOutputStream create(Path f) throws IOException;
```

它创建一个 Path 类代表的文件，并返回一个输出流。这个方法有多个重载方法，可以用来设置是否覆盖已有文件、该文件复制的份数、写入时的缓冲区大小、文件块大小(block)、权限等。默认情况下，如果 Path 中文件的父目录(或者更上一级目录)不存在，这些目录会被自动创建。

2. 读取数据

通过调用 FileSystem 实例的 open 方法打开文件，得到一个输入流。下面是使用 FileSystem 类读取 HDFS 中文件内容的完整程序：

```
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
public class FileSystemCat {
    public static void main(String[] args) throws Exception {
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);
        FSDataInputStream in = null;
        try {
            in = fs.open(new Path(uri));
            IOUtils.copyBytes(in, System.out, 4096, false);
        } finally {
            IOUtils.closeStream(in);
        }
    }
}
```

此外，FSDataInputStream 类同时也实现了 PositionedReadable(org.apache.hadoop.fs.PositionedReadable)接口，接口中定义的三个方法允许在任意位置读取文件的内容：

```
public int read(long position, byte[] buffer, int offset, int length) throws IOException;
public void readFully(long position, byte[] buffer, int offset, int length)
  throws IOException;
public void readFully(long position, byte[] buffer) throws IOException;
```

结合第 2 章内容，下面我们结合程序实现深入剖析 HDFS 读文件时的数据流向过程。

(1) 客户端通过调用 FileSystem.open()方法打开一个文件，对于 HDFS 来讲，其实是调用 DistributedFileSystem 实例的 open 方法。

(2) DistributedFileSystem 通过远程方法调用访问 NameNode，获取该文件的前几个

blocks 所在的位置信息；针对每个 block，NameNode 都会返回有该 block 数据信息的所有 DataNodes 节点，比如配置的 dfs.replication 为 3，就会每个 block 返回 3 个 DataNodes 节点信息，这些节点是按距离客户端的远近排序的，如果发起读文件的客户端就在包含该 block 的 DataNode 上，那么这个 DataNode 就排第一位(这种情况在做 Map 任务时常见)，客户端就会从本机读取数据。

DistributedFileSystem 的 open 方法返回一个 FSDataInputStream，FSDataInputStream 里包装着一个 DFSInputStream，DFSInputStream 真正管理 DataNodes 和 NameNode 的 I/O。

(3) 客户端调用 FSDataInputStream.read()方法，FSDataInputStream 里已经缓存了该文件前几个 block 所在的 DataNode 的地址，于是，从第一个 block 的第一个地址(也就是最近的 DataNode)开始连接读取。

(4) 反复调用 read()方法，数据不断地从 DataNode 流向客户端。

(5) 当一个 block 的数据读完时，DFSInputStream 会关闭当前 DataNode 的连接，打开下一个 block 所在的最优 DataNode 的连接继续读取；这些对客户端是透明的，在客户端看来，就是在读一个连续的流。

(6) 这样，一个 block 一个 block 地读下去，当需要使用更多 block 的存储信息时，DFSInputStream 会再次调用 NameNode，获取下一批 block 的存储位置信息，直到客户端停止读取，调用 FSDataInputStream.close()方法，整个读取过程结束。

小提示

文件操作还可以使用 Hadoop URL 的方式，示例代码如下：

```
import java.io.InputStream;
import java.net.URL;
import org.apache.hadoop.fs.FsUrlStreamHandlerFactory;
import org.apache.hadoop.io.IOUtils;

public class URLCat {
    static {
        URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
    }
    public static void main(String[] args) throws Exception {
        InputStream in = null;
        try {
            in = new URL(args[0]).openStream();
            IOUtils.copyBytes(in, System.out, 4096, false);
        } finally {
            IOUtils.closeStream(in);
        }
    }
}
```

在上面的程序中，先设置 URLStreamHandlerFactory，然后通过 URL 打开一个流，读取流，就得到了文件的内容，通过 IOUtils.copyBytes()把读到的内容写出到标准输出流里，也就是控制台上，从而实现了类似于 Linux 里的 cat 命令的功能。最后关闭输入流。

3. 写入数据

与读操作类似，Hadoop 对于写操作也提供了一个类：FSDataOutputStream，这个类重载了很多 java.io.DataOutputStream 的 write 方法，用于写入很多类型的数据，比如 int、char、字节数组等。

HDFS 写文件的示例代码如下：

```
FileSystem hdfs = FileSystem.get(new Configuration());
Path path = new Path("/testfile");

FSDataOutputStream dos = hdfs.create(path);
byte[] readBuf = "Hello World".getBytes("UTF-8");
dos.write(readBuf, 0, readBuf.length);
dos.close();
hdfs.close();
```

如果希望向已有文件追加内容，可以调用：

```
public FSDataOutputStream append(Path f) throws IOException;
```

如果文件不存在时，append 方法也可以用来新建一个文件。

下面，我们结合以上的程序，深入剖析 HDFS 写文件时的数据流向过程。

(1) 客户端调用 DistributedFileSystem.create()方法创建一个文件。

(2) DistributedFileSystem 向 NameNode 发起远程方法调用，创建一个文件，但是，NameNode 没有把它关联到任何 block 上去；NameNode 在这一步做了很多检查工作，保证该文件当前不存在，客户端有创建该文件的权限等。如果这些检查都通过了，NameNode 创建一条新文件记录；否则，创建失败，客户端返回 IOException。DistributedFileSystem 返回一个 FSDataOutputStream，像读文件时一样，这个 FSDataOutputStream 里包装着一个 DFSOutputStream，由它来实际处理与 DataNodes 和 NameNode 的通信。

(3) 客户端向 DFSOutputStream 里写数据，DFSOutputStream 把数据分成包，丢进一个称为 data queue 的队列中。DataStreamer 负责向 NameNode 申请新的 block，新的 block 被分配在了一个或多个(默认为 3 个)节点上，这些节点就形成一个管道。

(4) DataStreamer 把 data queue 里的包拿出来，通过管道输送给第 1 个节点，第 1 个节点再通过管道输送给第 2 个节点，第 2 个再输送给第 3 个。以此类推。

(5) DFSOutputStream 同时还在内部维护一个通知队列，名叫 ack queue，里面保存发过的数据包。一个包只有被所有管道上的 DataNodes 通知收到了，才会被移除。如果任意一个 DataNode 接收失败了，首先，管道关闭，然后把 ack queue 里的包都放回到 data queue 的头部，以便使失败节点的下游节点不会丢失这些数据。打开管道，把坏节点移除，数据会继续向其他好节点输送，直到管道上的节点都完成了。如果少复制了一个节点，向 NameNode 报告一下，说现在这个 block 没有达到设定的副本数，然后就返回成功了，后期，NameNode 会组织一个异步的任务，把副本数恢复到设定值。然后，接下来的数据包和数据块正常写入。

如果多个 DataNodes 都失败了，会检测 hdfs-site.xml 里的 dfs.replication.min 参数，默认值是 1，意思是只要有 1 个 DataNode 接收成功，就认为数据写入成功了。客户端就会收到写入成功的返回。后期，Hadoop 会发起异步任务把副本数恢复到 dfs.replication 设置的值。

以上操作对客户端都是透明的，客户端不知道发生了这些事情，只知道写文件成功了。

(6) 当客户端完成数据写入后，调用流的 close()方法，这个操作把 data queue 里的所有剩余的包都发给管道。

(7) 等所有包都收到了写成功的反馈后，客户端通知 NameNode 写文件完成了。因为 DataStream 写文件前就先向 NameNode 申请 block 的位置信息了，所以写文件完成时，NameNode 已知道每个 block 的位置信息，它只需等最小的副本数写成功，就可以返回成功。

4. 文件读写位置

读取文件时(FSDataInputStream)，允许使用 seek()方法在文件中定位。支持随机访问，理论上，可以从流的任何位置读取数据，但调用 seek()方法的开销是相当巨大的，应该尽量少调用，尽可能地使程序做到顺序读。

由于 HDFS 只允许对一个打开的文件顺序写入，或向一个已有文件的尾部追加，不允许在任意位置写，FSDataOutputStream 没有 seek 方法。但 FSDataOutputStream 类提供了一个 getPos()方法，可以查询当前在往文件的哪个位置写的写入偏移量：

```
public long getPos() throws IOException;
```

5. 重命名

通过 FileSystem.rename()方法，可为指定的 HDFS 文件重命名：

```
protected void rename(Path src, Path dst, Options) throws IOException;
```

示例代码实现如下：

```
Configuration conf = new Configuration();
FileSystem hdfs = FileSystem.get(conf);
Path frpath = new Path("/test");    //旧的文件名
Path topath = new Path("/testNew");    //新的文件名
boolean isRename = hdfs.rename(frpath, topath);
String result = isRename? "成功" : "失败";
```

6. 删除操作

通过 FileSystem.delete()方法删除指定的 HDFS 文件或目录(永久删除)：

```
public boolean delete(Path f, boolean recursive) throws IOException;
```

其中，f 为需要删除文件的完整路径，recursive 用来确定是否进行递归删除。如果 f 是一个文件或空目录，则不论 recursive 是何值，都删除。如果 f 是一个非空目录，则 recursive 为 true 时，目录下内容全部删除；如果 recursive 为 false，不删除，并抛出 IOException。

示例代码实现如下：

```
Path f = new Path(fileName);
boolean isExists = hdfs.exists(f);
if (isExists) { //if exists, delete
   boolean isDel = hdfs.delete(f, true);
   System.out.println(fileName + "  delete? \t" + isDel);
} else {
   System.out.println(fileName + "  exist? \t" + isExists);
}
```

7. 文件夹操作

FileSystem 中创建文件夹的方法如下：

```
public boolean mkdirs(Path f) throws IOException;
```

与 java.io.File.mkdirs 方法一样，创建目录的同时，默认地创建缺失的父目录。我们一般不需要创建目录，一般在创建文件时，默认地就把所需的目录都创建好了。

目录创建的示例代码实现如下：

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
Path srcPath = new Path(path);
boolean isok = fs.mkdirs(srcPath);
if(isok) {
   System.out.println("create dir ok!");
} else {
   System.out.println("create dir failure");
}
fs.close();
```

使用 FileSystem 的 listStatus()方法能够列出某个目录中的所有文件：

```
public FileStatus[] listStatus(Path f) throws IOException
public FileStatus[] listStatus(Path f, PathFilter filter) throws IOException
public FileStatus[] listStatus(Path[] files) throws IOException
public FileStatus[] listStatus(Path[] files, PathFilter filter)
  throws IOException
```

这一组方法都接收 Path 参数，如果 Path 是一个文件，返回值是一个数组，数组里只有一个元素，是这个 Path 代表的文件的 FileStatus 对象；如果 Path 是一个目录，返回值数组是该目录下的所有文件和目录的 FileStatus 组成的数组，有可能是一个 0 长数组；如果参数是 Path[]，则返回值相当于多次调用单 Path，然后把返回值整合到一个数组里；如果参数中包含 PathFilter，则 PathFilter 会对返回的文件或目录进行过滤，返回满足条件的文件或目录，条件由开发者自行定义。

FileSystem 的 globStatus 方法利用通配符来列出文件和目录：

```
public FileStatus[] globStatus(Path pathPattern) throws IOException;
public FileStatus[] globStatus(Path pathPattern, PathFilter filter)
  throws IOException;
```

文件夹删除操作与文件删除类似。

其他关于文件夹的操作方法还有 FileSystem.getWorkingDirectory(返回当前工作目录)、FileSystem.setWorkingDirectory(更改当前工作目录)等。

8. 属性操作

FileSystem 类中的 getFileStatus()方法返回一个 FileStatus 实例，该 FileStatus 实例中，包含了该 Path(文件或目录)的元数据信息：文件大小、block 大小、复制的份数、最后修改时间、所有者、权限等。示例代码实现如下：

```
FileStatus status = fs.getFileStatus(path);
System.out.println("path = " + status.getPath());
System.out.println("owner = " + status.getOwner());
```

```
System.out.println("block size = " + status.getBlockSize());
System.out.println("permission = " + status.getPermission());
System.out.println("replication = " + status.getReplication());
```

3.1.3 WebHDFS

Hadoop 提供的 Java Native API 支持对文件或目录的操作，为开发者提供了极大的便利。为满足许多外部应用程序操作 HDFS 文件系统的需求，Hadoop 提供了两种基于 HTTP 方式的接口：一是用于浏览文件系统的 Web 界面；另一个是 WebHDFS REST API 接口。

启动 HDFS 时，NameNode 和 DataNode 各自启动了一个内置的 Web 服务器，显示了集群当前的基本状态和信息。默认配置下 NameNode 的首页地址是 http://namenode-name:50070/。这个页面列出了集群里的所有 DataNode 和集群的基本状态。

这个 Web 界面也可以用来浏览整个文件系统。使用 NameNode 首页上的 Browse the file system 链接，输入需要查看的目录地址，即可看到，如图 3-7 所示。

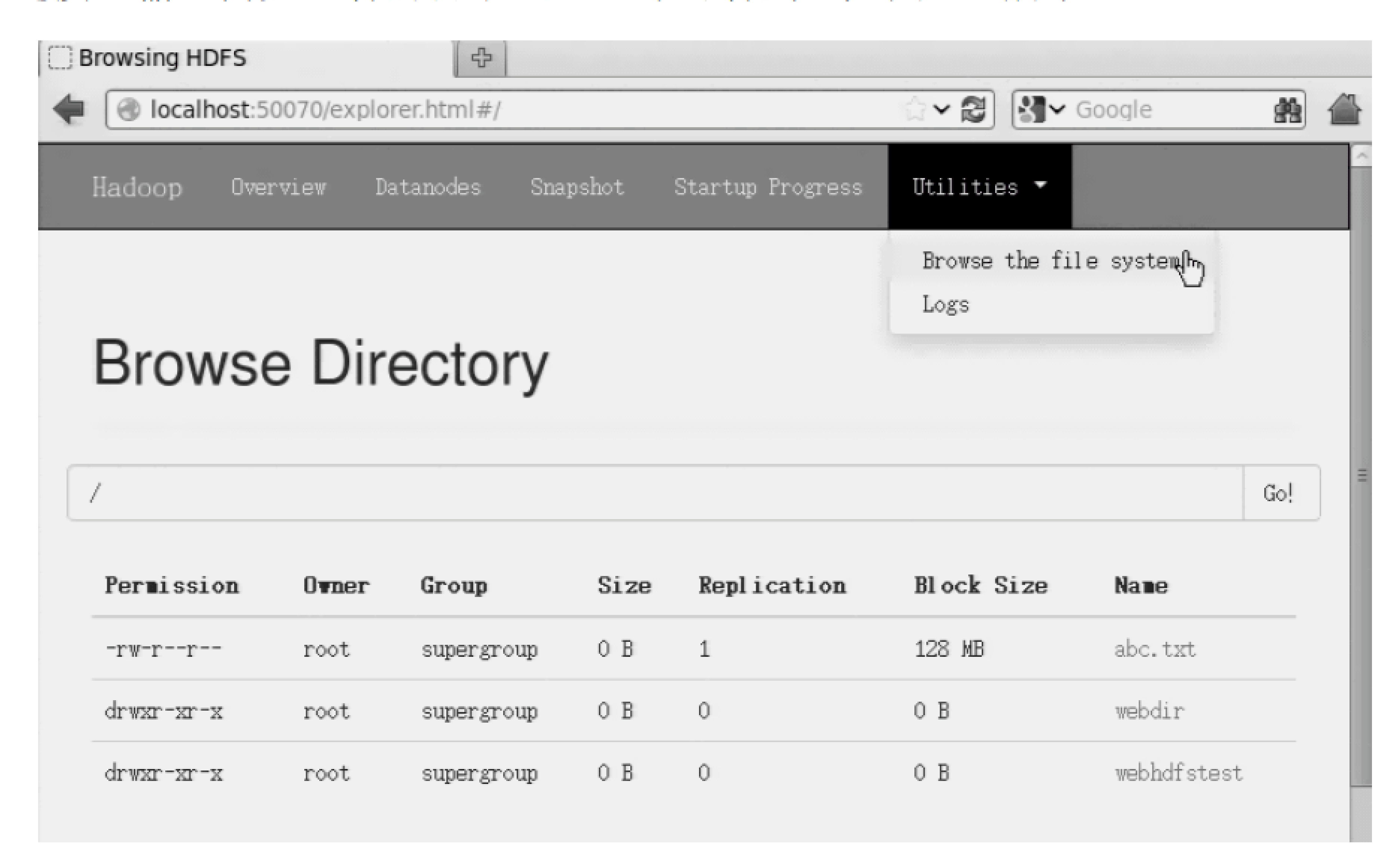


图 3-7　Web 界面

WebHDFS 基于 HTTP，通过 GET、PUT、POST 和 DELETE 等操作，支持 FileSystem/FileContext 的全部 API。具体操作类型见表 3-11。

表 3-11　WebHDFS 的操作

HTTP 方法	操作命令	对应的 FileSystem 接口
HTTP GET	OPEN	FileSystem.open
	GETFILESTATUS	FileSystem.getFileStatus
	LISTSTATUS	FileSystem.listStatus
	GETCONTENTSUMMARY	FileSystem.getContentSummary
	GETFILECHECKSUM	FileSystem.getFileChecksum
	GETHOMEDIRECTORY	FileSystem.getHomeDirectory

续表

HTTP 方法	操作命令	对应的 FileSystem 接口
HTTP GET	GETDELEGATIONTOKEN	FileSystem.getDelegationToken
	GETDELEGATIONTOKENS	FileSystem.getDelegationTokens
	GETXATTR	FileSystem.getXAttr
	GETXATTRS	FileSystem.getXAttrs
	LISTXATTRS	FileSystem.listXAttrs
	CHECKACCESS	FileSystem.access
HTTP PUT	CREATE	FileSystem.create
	MKDIRS	FileSystem.mkdirs
	CREATESYMLINK	FileContext.createSymlink
	RENAME	FileSystem.rename
	SETREPLICATION	FileSystem.setReplication
	SETOWNER	FileSystem.setOwner
	SETPERMISSION	FileSystem.setPermission
	SETTIMES	FileSystem.setTimes
	RENEWDELEGATIONTOKEN	FileSystem.renewDelegationToken
	CANCELDELEGATIONTOKEN	FileSystem.cancelDelegationToken
	CREATESNAPSHOT	FileSystem.createSnapshot
	RENAMESNAPSHOT	FileSystem.renameSnapshot
	SETXATTR	FileSystem.setXAttr
	REMOVEXATTR	FileSystem.removeXAttr
HTTP POST	APPEND	FileSystem.append
	CONCAT	FileSystem.concat
HTTP DELETE	DELETE	FileSystem.delete
	DELETESNAPSHOT	FileSystem.deleteSnapshot

在使用 WebHDFS REST API 接口前，要先对 Hadoop 进行配置和授权认证。编辑 hdfs-site.xml 文件，添加启用 WebHDFS(dfs.webhdfs.enabled)、kerberos 验证(dfs.web.authentication.kerberos.principal、dfs.web.authentication.kerberos.keytab)等属性配置。配置完成后，启动 WebHDFS 服务即可，如图 3-8 所示。

WebHDFS 默认的 HTTP 服务端口是 14000。需要说明的是，WebHDFS 的 FileSystem 模式是“webhdfs://”，URI 的格式如下：

```
webhdfs://<HOST>:<HTTP_PORT>/<PATH>
```

与之对应的 HDFS URI 格式如下：

```
hdfs://<HOST>:<RPC_PORT>/<PATH>
```

```
[root@test sbin]# ./httpfs.sh start

Setting HTTPFS_HOME:          /root/hadoop/hadoop
Setting HTTPFS_CONFIG:        /root/hadoop/hadoop/etc/hadoop
Sourcing:                    /root/hadoop/hadoop/etc/hadoop/httpfs-env.sh
Setting HTTPFS_LOG:           /root/hadoop/hadoop/logs
Setting HTTPFS_TEMP:           /root/hadoop/hadoop/temp
Setting HTTPFS_HTTP_PORT:     14000
Setting HTTPFS_ADMIN_PORT:     14001
Setting HTTPFS_HTTP_HOSTNAME:  test.hadoop
Setting HTTPFS_SSL_ENABLED:  false
Setting HTTPFS_SSL_KEYSTORE_FILE:      /root/.keystore
Setting HTTPFS_SSL_KEYSTORE_PASS:      password
Setting CATALINA_BASE:       /root/hadoop/hadoop/share/hadoop/httpfs/tomcat
Setting HTTPFS_CATALINA_HOME:       /root/hadoop/hadoop/share/hadoop/httpfs/tomcat
Setting CATALINA_OUT:        /root/hadoop/hadoop/logs/httpfs-catalina.out
Setting CATALINA_PID:        /tmp/httpfs.pid
```

图 3-8　启动 WebHDFS 服务

在 REST API 接口中，在 path 之前插入前缀“/webhdfs/v1”，操作语句被追加到最后，相应的 HTTP URL 格式如下：

```
http://<HOST>:<HTTP_PORT>/webhdfs/v1/<PATH>?op=...
```

下面我们以具体实例，来测试一下 WebHDFS 的功能。使用 curl 命令工具在 HDFS 根目录下创建一个名为“webdir”的目录，如图 3-9 所示。

```
[root@test hadoop]# hdfs dfs -ls /
Found 1 items
drwxr-xr-x   - root supergroup          0 2016-03-10 04:42 /webhdfstest
[root@test hadoop]# curl -i -X PUT "http://test.hadoop:14000/webhdfs/v1/webdir?user.name=root&op=
MKDIRS"
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: hadoop.auth="u=root&p=root&t=simple&e=1457592222388&s=5pE/nmfBgmkzgV2DqYOVAiteJ1s=";
Path=/; Expires=   , 10-   -2016 06:43:42 GMT; HttpOnly
Content-Type: application/json
Transfer-Encoding: chunked
Date: Wed, 09 Mar 2016 20:43:42 GMT

{"boolean":true}
[root@test hadoop]# hdfs dfs -ls /
Found 2 items
drwxr-xr-x   - root supergroup          0 2016-03-10 04:43 /webdir
drwxr-xr-x   - root supergroup          0 2016-03-10 04:42 /webhdfstest
[root@test hadoop]#
```

图 3-9　WebHDFS 创建目录的运行结果

3.1.4　其他接口

HDFS 支持的使用接口除了前面介绍过的 Java 等以外，还有 C、Thrift、HttpFS、HFTP、NFS 等。下面简单介绍几种。

1. C 接口

HDFS 基于 Java 编写，并没有提供原生的 C 语言访问接口，但 HDFS 提供了基于 JNI(Java Native Interface)的 C 调用接口 libhdfs，使 C 语言访问 HDFS 成为可能。

libhdfs 接口的头文件和库文件已包含在 Hadoop 发行版本中，可以直接使用。它的头文件 hdfs.h 一般位于{HADOOP_HOME}/include 目录中，而其库文件 libhdfs.so 通常则位于

{HADOOP_HOME}/lib/native 目录中。不同的版本，库文件所在位置稍有不同。

通过 libhdfs 访问 HDFS 文件系统与使用 C 语言 API 访问普通操作系统的文件系统类似。C++访问 HDFS 的方式也与 C 语言类似。接口主要如下。

(1) 建立、关闭与 HDFS 连接：hdfsConnect()、hdfsConnectAsUser()、hdfsDisconnect()。

(2) 打开、关闭 HDFS 文件：hdfsOpenFile()、hdfsCloseFile()。当用 hdfsOpenFile()创建文件时，可以指定 replication 和 blocksize 参数。

(3) 读 HDFS 文件：hdfsRead()、hdfsPread()。

(4) 写 HDFS 文件：hdfsWrite()。HDFS 不支持随机写，只能是从文件头顺序写入。

(5) 查询 HDFS 文件信息：hdfsGetPathInfo()。

(6) 查询数据块所在节点信息：hdfsGetHosts()。返回一个或多个数据块所在数据节点的信息，一个数据块可能存在于多个数据节点上。

libhdfs 中的函数是通过 JNI 调用 Java 虚拟机的，在虚拟机中构造对应的 HDFS 的 Java 类，然后反射调用该类的功能函数，占用内存较多，不适合对虚拟要求较高的场景。

下面是一个简单的例子：

```
#include "hdfs.h"

int main(int argc, char **argv) {

    hdfsFS fs = hdfsConnect("default", 0);
    const char *writePath = "/tmp/testfile.txt";
    hdfsFile writeFile =
      hdfsOpenFile(fs, writePath, O_WRONLY|O_CREAT, 0, 0, 0);
    if(!writeFile) {
       fprintf(stderr, "Failed to open %s for writing!\n", writePath);
       exit(-1);
    }
    char *buffer = "Hello, World!";
    tSize num_written_bytes =
      hdfsWrite(fs, writeFile, (void*)buffer, strlen(buffer)+1);
    if (hdfsFlush(fs, writeFile)) {
       fprintf(stderr, "Failed to 'flush' %s\n", writePath);
       exit(-1);
    }
    hdfsCloseFile(fs, writeFile);
}
```

2. HFTP

HFTP 是一个可以实现从远程 HDFS 集群读取 Hadoop 文件系统数据的接口。HFTP 默认是打开的，数据读取通过 HTTP 协议，允许以浏览器的方式访问和下载所有文件。这种方式带来便利的同时，也存在一定的安全隐患。

HFTP 是一个只读的文件系统，如果试图用它写或者修改文件系统的状态，将会抛出一个错误。如果使用多个不同版本的 HDFS 集群时，需要在集群之间移动数据，HFTP 是非常有用的。HFTP 在不同 HDFS 版本之间都是兼容的，通常与 distcp 结合使用实现并行复制。

HSFTP 是 HFTP 的一个扩展，默认使用 HTTPS 在传输时加密数据。

3. HttpFS

HttpFS 是 Cloudera 公司提供的一个 Web 应用，一般部署在内嵌的 Web 服务器中，但

独立于 Hadoop 的 NameNode。

HttpFS 是提供 REST HTTP 接口的服务器，可以支持全部 HDFS 文件系统操作。通过 WebHDFS REST API，可以对 HDFS 进行读写等访问操作。与 WebHDFS 的区别是，不需要客户端，就可以访问 Hadoop 集群的每一个节点。

通过 HttpFS，可以访问放置在防火墙后面的 Hadoop 集群数据。HttpFS 可以作为一个网关角色，是唯一可以穿过防火墙访问内部集群数据的系统。

HttpFS 的内置安全特性支持 Hadoop 伪身份验证和 HTTP SPNEGO Kerberos 及其他插件式(pluggable)验证机制。它还提供了对 Hadoop 代理用户的支持。

3.2　操 作 实 践

前面主要介绍了 HDFS 系统接口和编程方式，本节介绍 HDFS 中 Java 编程的操作实例。

3.2.1　文件操作

使用命令行编写 HDFS 程序，通常有三个步骤。

首先，编写 HDFS 程序源码，并通过 java 编译器编译成字节码。

然后，将字节码打包成 JAR 文件。

最后，通过 Hadoop 加载 JAR 文件，并运行。

下面，我们以一个完整的文件操作为例来说明。程序的主要功能如下。

(1)　在 HDFS 文件系统中创建一个名为“hdtest”的目录。

(2)　将本地名为“hfile.txt”的文件上传到 HDFS 中的 hdtest 目录下面。

(3)　遍历 hdtest 目录。

(4)　将 HDFS 中的 hdtest/hfile.txt 文件下载到本地，并另存为“hfile2.txt”。

程序的源代码如下：

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;

public class HdfsTest {
    private static final String HADOOP_URL = "hdfs://test.hadoop:9000";
    private Configuration conf;

    /**
     * 构造函数
     */
    public HdfsTest() {
        this.conf = new Configuration();
```

```
}
/**
 * 测试入口函数
 */
public static void main(String[] args) throws IOException {
    HdfsTest hdfs = new HdfsTest();
    hdfs.createDir("/hdtest"); //创建目录
    hdfs.copyFile("file/hfile.txt", "/hdtest/hfile.txt"); //拷贝文件
    hdfs.ls("/hdtest");  //遍历目录
    hdfs.cat("/hdtest/hfile.txt");  //查看文件内容
    //下载文件并另存
    hdfs.download("/hdtest/hfile.txt", "file/hfile2.txt");
}
 /**
 * 创建目录
 * @param folder
 * @throws IOException
 */
public void createDir(String folder) throws IOException {
    Path path = new Path(folder);
    FileSystem fs = FileSystem.get(conf);
    if (!fs.exists(path)) {
        fs.mkdirs(path);
        System.out.println("Create: " + folder);
    }
    fs.close();
}
/**
 * 上传文件到 HDFS
 * @param local
 * @param remote
 * @throws IOException
 */
public void copyFile(String local, String remote) throws IOException {
    FileSystem fs = FileSystem.get(conf);
    fs.copyFromLocalFile(new Path(local), new Path(remote));
    System.out.println("copy from: " + local + " to " + remote);
    fs.close();
}
/**
 * 遍历文件
 * @param folder
 * @throws IOException
 */
public void ls(String folder) throws IOException {
    Path path = new Path(folder);
    FileSystem fs = FileSystem.get(conf);
    FileStatus[] list = fs.listStatus(path);
    System.out.println("ls: " + folder);
    System.out.println("**********list begin*************");
    for (FileStatus f : list) {
        System.out.printf("name: %s, folder: %s, size: %d\n",
                          f.getPath(), f.isDir(), f.getLen());
    }
    System.out.println("**********list end*************");
    fs.close();
}
/**
 * 查看文件中的内容
 * @param remoteFile
 * @return
```

```
     * @throws IOException
     */
    public String cat(String remoteFile) throws IOException {
        Path path = new Path(remoteFile);
        FileSystem fs = FileSystem.get(conf);
        FSDataInputStream fsdis = null;
        System.out.println("Content: " + remoteFile);

        OutputStream baos = new ByteArrayOutputStream();
        String str = null;
        try {
            fsdis = fs.open(path);
            IOUtils.copyBytes(fsdis, baos, 4096, false);
            str = baos.toString();
        } finally {
            IOUtils.closeStream(fsdis);
            fs.close();
        }
        System.out.println(str);
        return str;
    }
    /**
     * 从 HDFS 中下载文件到本地
     * @param remote
     * @param local
     * @throws IOException
     */
    public void download(String remote, String local) throws IOException {
        Path path = new Path(remote);
        FileSystem fs = FileSystem.get(conf);
        fs.copyToLocalFile(path, new Path(local));
        System.out.println(
          "download file from'" + remote + "' to '" + local + "'");
        fs.close();
    }
    /**
     * 重命名文件
     * @param src
     * @param dst
     * @throws IOException
     */
    public void rename(String src, String dst) throws IOException {
        FileSystem fs = FileSystem.get(conf);
        fs.rename(new Path(src), new Path(dst));
        System.out.println("Rename:  " + src + " to " + dst);
        fs.close();
    }
    /**
     * 删除文件或目录
     * @param folder
     * @throws IOException
     */
    public void delete(String folder) throws IOException {
        Path path = new Path(folder);
        FileSystem fs = FileSystem.get(conf);
        fs.deleteOnExit(path);
        System.out.println("Delete: " + folder);
        fs.close();
    }
}
```

编译 HdfsTest.java 源文件。Hadoop 2.x 版本中 JAR 不再集中在一个 hadoop-core*.jar 中，而是分成多个 JAR(如$HADOOP_HOME/share/hadoop/common/hadoop-common-2.6.0.jar、$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.6.0.jar、$HADOOP_HOME/share/hadoop/common/lib/commons-cli-1.2.jar 等)，通过“hadoop classpath”命令，可以得到运行 Hadoop 程序所需的全部 classpath 信息。

我们将 Hadoop 的 classhpath 信息添加到 CLASSPATH 变量中，然后直接编译：

```
$ javac HdfsTest.java
```

编译时会有警告，可以忽略。编译后，可以看到生成的.class 文件，如图 3-10 所示。

```
[root@test class]# ls
file  HdfsTest.java
[root@test class]# javac HdfsTest.java
注: HdfsTest.java使用或覆盖了已过时的 API。
注: 有关详细信息, 请使用 -Xlint:deprecation 重新编译。
[root@test class]# ls
file  HdfsTest.class  HdfsTest.java
[root@test class]#
```

图 3-10　编译并查看生成的.class 文件

打包.class 文件，如图 3-11 所示。

```
[root@test class]# jar -cvf HdfsTest.jar *.class
已添加清单
正在添加: HdfsTest.class(输入 = 3620) (输出 = 1812)(压缩了 49%)
[root@test class]# ls
file  HdfsTest.class  HdfsTest.jar  HdfsTest.java
[root@test class]#
```

图 3-11　打包.class 文件并查看

运行测试，结果如图 3-12 所示。

```
[root@test class]# hdfs dfs -ls /
Found 2 items
-rw-r--r--   1 root supergroup          0 2016-03-10 23:24 /abc.txt
drwxr-xr-x   - root supergroup          0 2016-03-10 04:43 /webdir
[root@test class]# hadoop jar HdfsTest.jar HdfsTest
Create: /hdtest
copy from: file/hfile.txt to /hdtest/hfile.txt
ls: /hdtest
**********list begin*************
name: hdfs://test.hadoop:9000/hdtest/hfile.txt, folder: false, size: 29
**********list end*************
Content: /hdtest/hfile.txt
Hi, I'm a test file on HDFS.

download file from'/hdtest/hfile.txt' to 'file/hfile2.txt'
[root@test class]# hdfs dfs -ls /
Found 3 items
-rw-r--r--   1 root supergroup          0 2016-03-10 23:24 /abc.txt
drwxr-xr-x   - root supergroup          0 2016-03-17 04:39 /hdtest
drwxr-xr-x   - root supergroup          0 2016-03-10 04:43 /webdir
[root@test class]#
```

图 3-12　运行测试结果

由上面的运行结果可以看到，我们在 HDFS 文件系统中成功地创建了目录并上传/下载了一个文件。通过 Fs Shell 命令，可以验证查看已上传的文件，如图 3-13 所示。

```
[root@test class]# hdfs dfs -ls /hdtest
Found 1 items
-rw-r--r--   1 root supergroup         29 2016-03-17 04:39 /hdtest/hfile.txt
[root@test class]# hdfs dfs -cat /hdtest/hfile.txt
Hi, I'm a test file on HDFS.
[root@test class]#
```

图 3-13　验证查看已上传的文件

此外，上述实例代码中，还提供了重命名(rename)和删除(delete)函数，感兴趣的读者可以自己测试一下。

使用命令行编译运行 Java 程序有些麻烦，每修改一次就需要手动编译、打包一次。对于较大规模的应用，可以使用 Eclipse 等集成环境进行开发，以提高开发效率。

3.2.2　压缩与解压缩

我们在 HDFS 中对数据进行压缩处理来优化磁盘使用率，提高数据在磁盘和网络中的传输速度，从而提高系统处理数据的效率。

Hadoop 应对压缩格式的技术是自动识别。如果我们压缩的文件有相应压缩格式的扩展名(比如 lzo、gz、bzip2 等)，Hadoop 会根据压缩格式的扩展名，自动选择相对应的解码器来解压数据，此过程完全是 Hadoop 自动处理的，我们只须确保输入的压缩文件有扩展名。

Hadoop 在 Codec 类(org.apache.hadoop.io.compress)中，实现了压缩和解压缩的接口 CompressionCodec。可用的 Codec 实现类见表 3-12。

表 3-12　可用的 Codec 实现类

压缩格式	Codec 实现类
Deflate	org.apache.hadoop.io.compress.DefaultCodec
Gzip	org.apache.hadoop.io.compress.GzipCodec
Bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
LZ4	org.apache.hadoop.io.compress.Lz4Codec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

CompressionCodec 有两个方法，可以帮助我们方便地压缩或解压数据。压缩数据时使用 createOutputStream(OutputStream out)获取压缩输出流对象 CompressionOutputStream，我们将未压缩的数据写入该流，它会帮我们压缩数据后，写出至底层的数据流 out。

相反地，在解析数据的时候，使用 createInputStream(InputStream in)获取解压缩输入流对象 CompressionInputstream，通过它，我们可以从底层的数据流中读取解压后的数据。

CompressionOutputStream、CompressionInputStream 与 java.util.zip.DeflaterOutputStream、java.util.zip.DeflaterInputStream 类似，但是，前者支持重置内部的压缩器(Compressor)与解压缩器(Decompressor)状态。

CompressionCodecFactory 是 Hadoop 压缩框架中的另一个类，主要功能是负责根据不同的文件扩展名，来自动地获取相对应的压缩解压器，使用者可以通过它提供的方法，获得 CompressionCodec，极大地增强了应用程序在处理压缩文件时的通用性。

除了前面介绍的 createInputStream()和 createInputStream()方法外，Hadoop 中还有其他两种压缩模式。

一是压缩机 Compressor 和解压机 Decompressor。在 Hadoop 的实现中，数据编码器和解码器被抽象成了两个接口：org.apache.hadoop.io.compress.Compressor 和 org.apache.hadoop.io.compress.Decompressor。它们规定了一系列的方法，所以，在 Hadoop 内部的编码/解码算法实现中都需要实现对应的接口。在实际的数据压缩与解压缩过程中，Hadoop 为用户提供了统一的 I/O 流处理模式。

二是压缩流 CompressionOutputStream 和解压缩流 CompressionInputStream。这两个类分别继承自 java.io.OutputStream 和 java.io.InputStream，作用也类似。

下面，我们编码实现文件的压缩和解压缩操作。源程序如下：

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.CompressionCodecFactory;
import org.apache.hadoop.io.compress.CompressionInputStream;
import org.apache.hadoop.io.compress.CompressionOutputStream;
import org.apache.hadoop.util.ReflectionUtils;
public class CompressTest {
    /**
     * 压缩文件
     * @param codecClassName
     * @param filein, fileout
     * @throws IOException
     */
    public static void compress(String codecClassName, String filein,
      String fileout) throws Exception {
        Class<?> codecClass = Class.forName(codecClassName);
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        CompressionCodec codec = (CompressionCodec)ReflectionUtils
                                .newInstance(codecClass, conf);
        //指定压缩文件路径
        FSDataOutputStream outputStream = fs.create(new Path(fileout));
        //指定要被压缩的文件路径
        FSDataInputStream in = fs.open(new Path(filein));
        //创建压缩输出流
        CompressionOutputStream out =
          codec.createOutputStream(outputStream);
        IOUtils.copyBytes(in, out, conf);
        IOUtils.closeStream(in);
        IOUtils.closeStream(out);
    }
    /**
     * 解压缩：使用文件扩展名来推断 codec
     * @param fileuri
     * @throws IOException
     */
```

```
    public static void uncompress(String fileuri) throws IOException {
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(fileuri), conf);
        Path inputPath = new Path(fileuri);
        CompressionCodecFactory factory = new CompressionCodecFactory(conf);
        CompressionCodec codec = factory.getCodec(inputPath);
        if(codec == null) {
            System.out.println("no codec for " + fileuri);
            System.exit(1);
        }
        String outputUri = CompressionCodecFactory.removeSuffix(
                          fileuri, codec.getDefaultExtension());
        InputStream in = null;
        OutputStream out = null;
        try {
            in = codec.createInputStream(fs.open(inputPath));
            out = fs.create(new Path(outputUri));
            IOUtils.copyBytes(in, out, conf);
        } finally {
            IOUtils.closeStream(out);
            IOUtils.closeStream(in);
        }
    }
    public static void main(String[] args) throws Exception {
        String filein = "/hdtest/bigdata.pdf";
        String fileout = "/hdtest/bigdatacom.gz";
        compress("org.apache.hadoop.io.compress.GzipCodec",
                 filein, fileout);
        //uncompress(fileout);
    }
}
```

编译并打包运行。压缩操作运行的结果如图 3-14 所示。

```
[root@test class]# hdfs dfs -ls /hdtest
Found 2 items
-rw-r--r--   1 root supergroup   71383576 2016-03-17 17:46 /hdtest/bigdata.pdf
-rw-r--r--   1 root supergroup         29 2016-03-17 04:39 /hdtest/hfile.txt
[root@test class]# hadoop jar CompressTest.jar CompressTest
16/03/17 17:50:04 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib library
16/03/17 17:50:04 INFO compress.CodecPool: Got brand-new compressor [.gz]
[root@test class]# hdfs dfs -ls /hdtest
Found 3 items
-rw-r--r--   1 root supergroup   71383576 2016-03-17 17:46 /hdtest/bigdata.pdf
-rw-r--r--   1 root supergroup   63638004 2016-03-17 17:50 /hdtest/bigdatacom.gz
-rw-r--r--   1 root supergroup         29 2016-03-17 04:39 /hdtest/hfile.txt
[root@test class]#
```

图 3-14　进行压缩操作并查看结果

解压缩操作的运行结果如图 3-15 所示。

```
[root@test class]# hdfs dfs -ls /hdtest
Found 2 items
-rw-r--r--   1 root supergroup   63638004 2016-03-17 17:50 /hdtest/bigdatacom.gz
-rw-r--r--   1 root supergroup         29 2016-03-17 04:39 /hdtest/hfile.txt
[root@test class]# hadoop jar CompressTest.jar CompressTest
16/03/17 17:53:26 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib library
16/03/17 17:53:26 INFO compress.CodecPool: Got brand-new decompressor [.gz]
[root@test class]# hdfs dfs -ls /hdtest
Found 3 items
-rw-r--r--   1 root supergroup   71383576 2016-03-17 17:53 /hdtest/bigdatacom
-rw-r--r--   1 root supergroup   63638004 2016-03-17 17:50 /hdtest/bigdatacom.gz
-rw-r--r--   1 root supergroup         29 2016-03-17 04:39 /hdtest/hfile.txt
[root@test class]#
```

图 3-15　解压缩的运行结果

3.3 小　　结

本章介绍了 HDFS 系统的操作使用和如何使用接口进行编程，通过诸多的实例，以实践的方式重点介绍了 Shell 命令接口、Java 编程接口、WebHDFS 等。

通过对本章内容的学习，读者应该能够熟练地掌握通过 Shell 接口操作 HDFS 文件系统的方式和方法，同时，能够使用 Java 编程语言、Web 等管理 HDFS 上的文件，为后续章节的大数据学习打下良好的基础。

大数据计算篇

第 4 章

YARN

学习目标

本章带领读者一起学习 YARN，它是 Hadoop 分布式集群中负责资源管理和调度的重要模块。为保证内容的权威性，本章主要以官方提供的资料为基础，再结合实践理解进行讲述，从 YARN 的概述、主要组成模块、调度器、RM 高可用、节点标签等方面对 YARN 展开详细的说明。讲述一个 Application 工作提交后，YARN 的工作原理和数据流程。在本章的最后，将给出编写 YARN 应用程序的实践案例。

通过本章的学习，读者应能够掌握 YARN 如何对资源进行管理和调度，能够掌握它的基本配置项，以及开启、关闭相关的功能。

本章要点

- YARN 概述及主要组成
- 容量、公平调度器
- RM 重启、高可用
- 节点标签
- YARN 服务注册
- YARN 应用编程

4.1 YARN 概述

Apache Hadoop YARN(Yet Another Resource Negotiator，另一种资源协调者)是从 Hadoop 0.23 进化来的一种新的资源管理和应用调度框架。基于 YARN，可以运行多种类型的应用程序，例如 MapReduce、Spark、Storm 等。YARN 不再具体管理应用，资源管理和应用管理是两个低耦合的模块。

YARN 从某种意义上来说，是一个云操作系统(Cloud OS)。基于该操作系统之上，程序员可以开发多种应用程序，例如批处理 MapReduce 程序、Spark 程序以及流式作业 Storm 程序等。这些应用可以同时利用 Hadoop 集群的数据资源和计算资源。此外，还可以利用 YARN 的资源管理器，提供新的应用管理器实现。

YARN 的产生是对 Hadoop1 的优化，这次优化是 Hadoop 框架发展以来最大的一次重构，Hadoop2 将 Jobtracter 的资源管理和任务管理拆分成两个独立模块，成为独立的 YARN 和各种 ApplicationManager。

概括地说，YARN 弥补了 Hadoop1 中的两个主要缺陷，这是产生 YARN 的原动力。

(1) 不支持多租赁。集群的计算资源只支持 MR 程序，不能支持 Storm、Spark 等作业。

(2) 规模限制，集群规模超过 4000 台时，会出现不可预测性，计算能力非近线性扩展。

YARN 的分布式系统，包含两个守护进程：资源管理器(ResourceManager，RM)、节点管理器(NodeManager，NM)。另外，很重要的进程是应用管理者(ApplicationManager，AM)，当一个应用提交后，RM 负责启动该任务对应类型的 AM，应用的执行由 AM 管理。可以简单地认为，AM 进程的生命周期为应用的运行时间。YARN 的工作原理如图 4-1 所示。

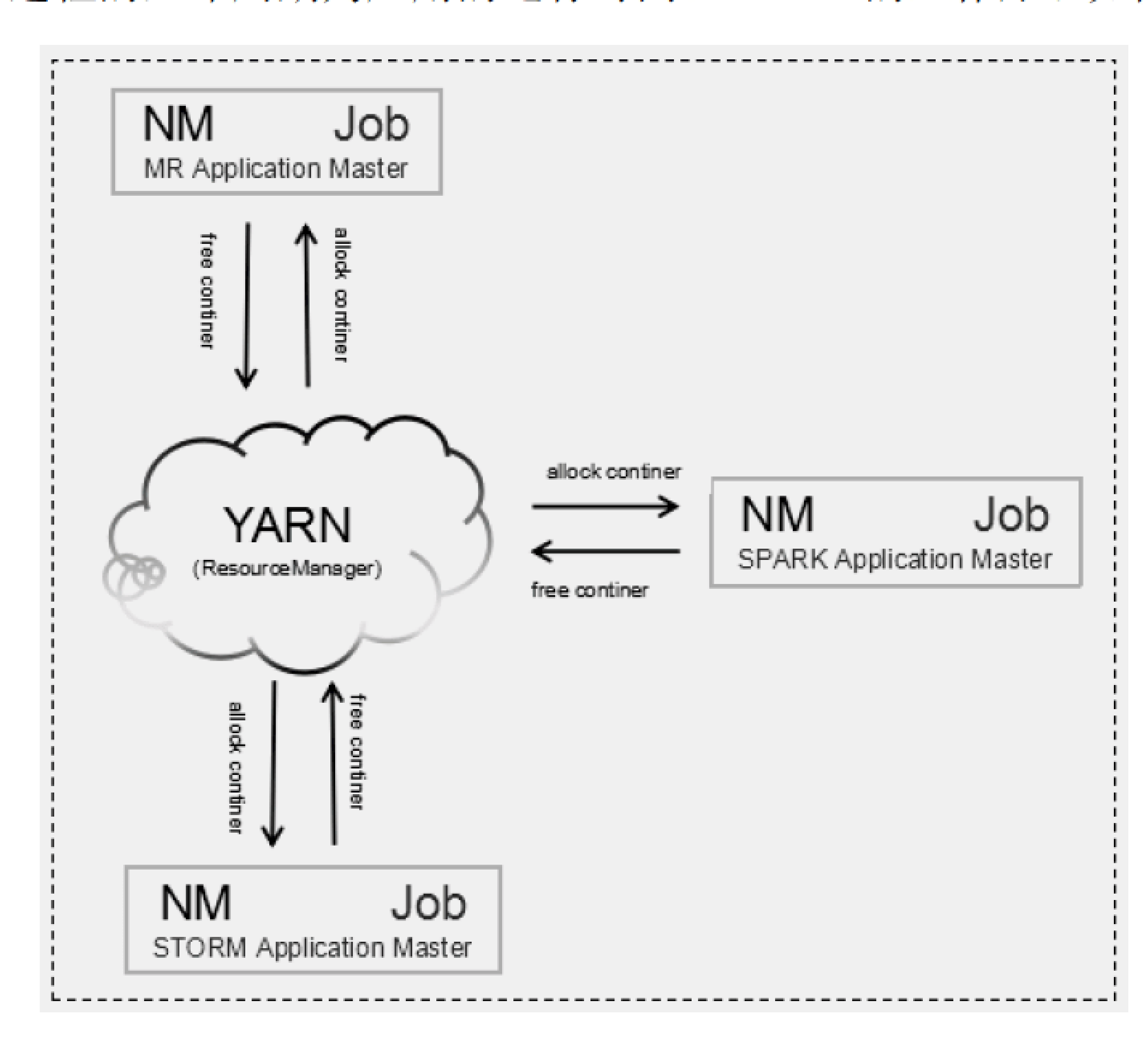


图 4-1 YARN 的工作原理

4.2 YARN 的主要组成模块

YARN 的主要模块包括如下几种。

(1) 资源管理器 ResourceManager(RM)：管理全部计算资源给各种应用，RM 和每个节点上的 NodeManager 构成资源管理和作业调度框架。

(2) 节点管理器 NodeManager(NM)：管理在该节点上的所有用户进程，并向 RM 汇报资源情况、用户进程情况。

(3) 应用管理器 ApplicationMaster(App Mstr 或 AM)：管理作业的启动和状态跟踪，对失败任务进行重试。每类应用的提交运行都要有对应的 App Mstr 来管理。

4.3 YARN 的整体设计

YARN 的设计初衷，是分离资源管理和作业调度/监控功能。核心思路是建立一个全局的资源管理器(RM)和多类型的作业管理器(AM)。

任何一个作业提交后(或者是单一的 job，或者是由多个组件构成的 DAG 有向无环图)，都有两个守护进程，分别是 RM 和 AM，两者构成了数据计算框架。

RM 具有对集群系统中所有作业进行资源分配的最高的权限。NM 是每个工作节点上的框架代理，主要负责容器开辟、监控容器的资源使用情况(CPU、内存、硬盘、带宽)，并报告使用信息给 RM。

每种类型的 AM 都是带有特定库的计算框架，AM 从 RM 申请计算资源，并与 NM 一起执行、监控作业。

RM 有两个重要的功能模块：调度器(Scheduler)和应用管理者(ApplicationsManager，注意不是 ApplicationMaster)。

调度器负责为各种作业程序分配资源。调度器只负责调度职责，不涉及监控或者跟踪作业状态的功能。调度器虽然负责作业调度，但它不负责失败作业的重启、失败任务的重启、硬件引起的失败故障。调度器根据作业的请求，以容器的形式进行调度，容器包含的元素有 CPU、内存、硬盘、带宽等。调度器选型可通过配置制定和变更，根据队列、作业对集群资源进行划分。

目前的调度器有容量调度器(CapacityScheduler)和公平调度器(FairScheduler)。

应用管理者(ApplicationsManager)负责接收提交作业，申请“第一容器”所需资源并提供服务，负责作业容器失败时的重启。每种类型应用的 AM 负责与调度器协定作业执行的资源容器，并跟踪作业状态、监控作业进度、故障处理等。

Hadoop 2.x 版本的 MapReduce API 兼容先前的 1.x 版本。这意味着所有在 Hadoop 1.x 下开发的 MapReduce job 程序，可仅重新编译而无修改地直接运行在 YARN 系统上。

从客户端提交任务到 RM 接受任务，再到 App Mstr 执行任务，基于 YARN 的任务工作流程如图 4-2 所示。

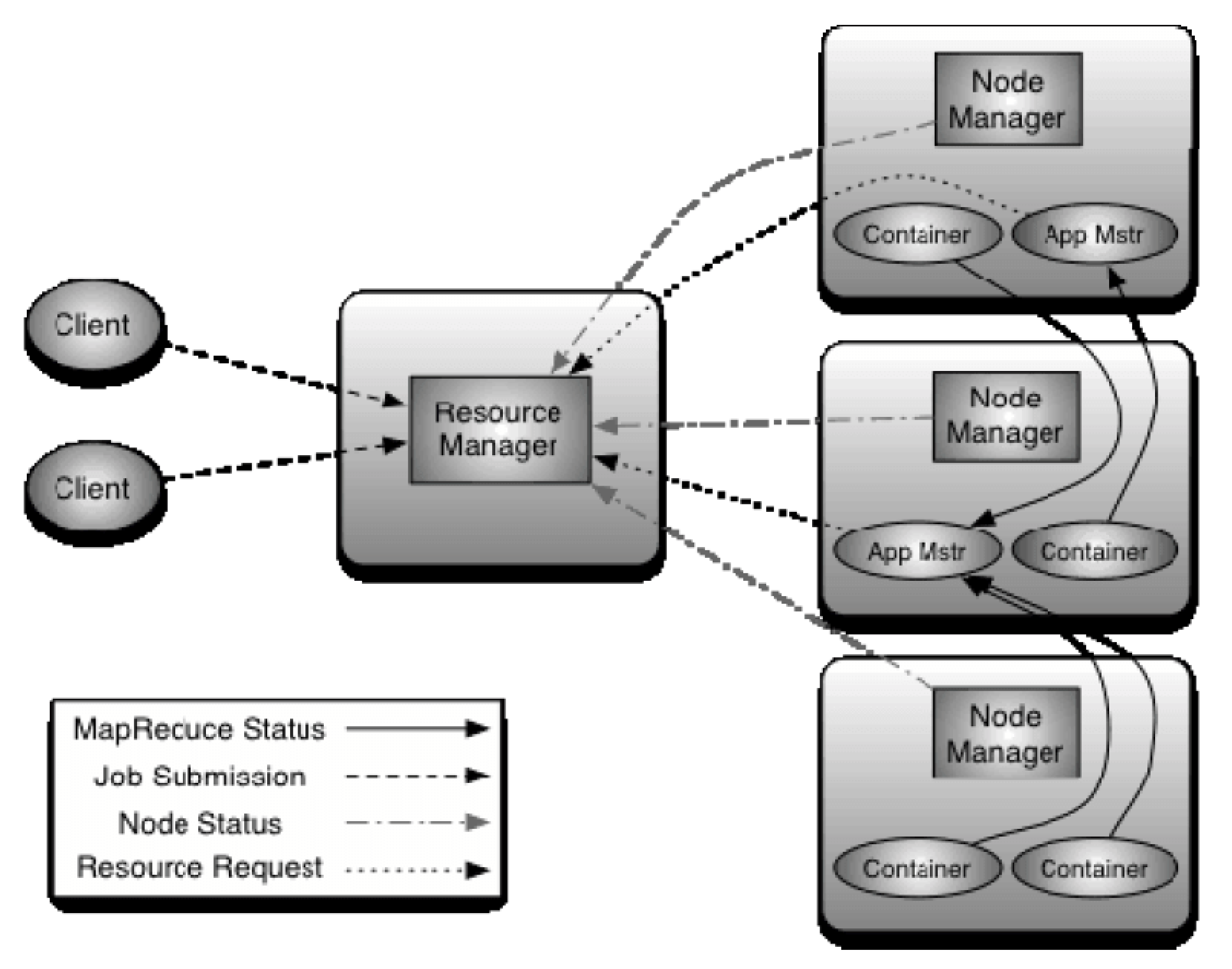


图 4-2　YARN 的工作流程

4.4　容量调度器

容量调度器用于灵活调度计算资源，最大化地利用共享计算资源。

4.4.1　什么是容量调度器

容量调度器是为实现集群的吞吐量和利用率最大化而设计的调度策略，Hadoop 作业运行时共享多租赁集群资源。一般来说，每个组织都拥有私属的计算资源，这些资源能够满足该组织所有业务在高峰时的计算需求。每个组织搭建独立集群的模式，会导致计算资源的不充分利用和对多个独立集群管理的高昂费用。很明显，在组织之间共享集群资源是一种高性价值比的模式，这样，可低成本重复获取经济效益而不用创建新的私有集群。

然而事实上，组织之间很关注集群的共享可能带来的负面影响，担心其他组织对共享资源占用严重而影响本部门的业务计算需求。

容量调度器是专门为解决上述问题而设计的集群计算资源调度器，它给每个组织一个计算资源的容量担保。容量调度器的核心思想是：Hadoop 集群的计算资源可以在多个组织间共享，并且，一个组织可以使用其他组织的空闲资源，这在成本效益方面提供了弹性机制。组织间共享集群，就需要支持多租赁模式，因为每一个组织必须有容量担保，以确保共享集群的每一个作业或用户或队列都能分配到合适的资源。容量调度器提供了一种严格的规则来保证每个作业、用户、队列非平均地消费集群资源，以使集群的利用率和吞吐量最大化。同时，容量调度器对同一用户或同一队列的初始化和特定作业的资源分配进行限制，以确保集群资源分配的公平性和稳定性。

容量调度器提出的基础概念是“队列”，“队列”由管理员设置，主要考量是共享集群的经济效益。为了提供更深入的控制和对共享资源的预见性，容量调度器支持多级队列，以确保一个队列的资源在各个子队列中被优先分配。在子队列都不再需要资源的情况下，其他队列才可以使用该队列的空闲计算资源。

4.4.2 容量调度器的特性

容量调度器作为 YARN 的默认调度器，具有普适性。下面，我们讲述容量调度器的特性有哪些。

(1) 多级队列：队列的分级，是为了确保队列资源被其他队列使用前有更多的控制和预见性。

(2) 容量担保：每个队列从资源总容量中申请的资源片段，用户提交作业到该队列，则使用该队列分配的资源。管理员可以对每个队列配置软限制和硬限制。

(3) 安全：每个队列有严格的访问控制列表。用户提交作业后，具有对该作业的访问、控制权限，但其他用户不能查看或修改该应作业。注意：每个队列和系统管理员可以查看、修改访问控制列表。

(4) 弹性：一个队列的空闲资源可以分配给其他队列的作业使用，即使该队列已经超出了配置的容量限制。队列中低于配置容量运行的作业对资源有更多需求时，其他队列的空闲资源将被分配过来。对集群中队列共享资源进行弹性分派，可以防止资源浪费，提高集群的利用率。

(5) 多租赁：防止单个作业、用户、队列独占集群资源，并确保集群资源不会过载。

(6) 操作性。

- 运行时配置：队列的定义和属性比如容量、ACLS，管理员可以在运行时修改。运行时修改队列的配置给用户造成的影响非常小。为管理员提供控制台来查看当前各个队列的分配情况。运行中，管理员可以增加新的队列，但是，不能删除队列。
- 关闭作业提交：管理员可以在运行时关闭队列提交功能，使得新作业不能被提交，以确保已开始作业运行完成。如队列的状态是 STOPPED，新应用不能提交到该队列及该队列的子队列，但是，已经存在的作业将会继续运行完成，当所有作业执行完成时，队列的关闭操作将自动结束，管理员也可以 START 已经处于 STOPPED 状态的队列。

(7) 基于资源的调度：对那些高资源需求的作业，可通过配置，来申请比默认高的资源，容量调度器可以满足对不同资源要求的作业。目前已支持对内存资源的需求管理。

(8) 基于用户或群组的队列映射：这个属性允许用户根据用户或群组映射 job 到一个指定的队列。

4.4.3 配置 RM 使用容量调度器

配置 conf/yarn-site.xml，用表 4-1 的属性指定调度器类型。

表 4-1　yarn-site.xml 配置信息

属　性	值
yarn.resourcemanager.scheduler.class	org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler

用户可以在 etc/hadoop/下看到 capacity-scheduler.xml，是专门配置容量调度器的配置文件。默认情况下，启动 default 队列，同时，还可配置一些其他参数，如：

```
<property>
   <name>yarn.scheduler.capacity.root.queues</name>
   <value>default</value>
   <description>
      The queues at the this level (root is the root queue).
   </description>
</property>
<property>
   <name>yarn.scheduler.capacity.maximum-applications</name>
   <value>10000</value>
   <description>
      Maximum number of applications that can be pending and running.
   </description>
</property>
<property>
   <name>
      yarn.scheduler.capacity.root.default.acl_submit_applications
   </name>
   <value>*</value>
   <description>
      The ACL of who can submit jobs to the default queue.
   </description>
</property>
```

4.5　公平调度器(Fair Scheduler)

公平调度器是对所有作业以平均、相等的分配策略对集群资源进行调度。

4.5.1　什么是公平调度器

YARN 可以调度多种资源类型，默认情况下，公平调度器仅是基于内存的公平调度。当单一作业运行时，该作业使用整个集群的资源，当其他作业提交时，空闲资源被分配给新作业，每个作业最终获取了相同的处理资源时间。不像 Hadoop 默认的容量调度器是根据队列分配资源，公平调度策略能使短任务在合理时间内完成，而不需要等待较长的时间，这是在不同用户之间共享资源的合理方法。公平策略可带优先级属性，优先级的权重表示每个作业从整体资源中获取资源的数量。

公平调度器将所有的作业划分到不同的队列中，这些队列公平占有资源。默认情况下，所有用户都在一个队列中，这个队列的名称是 default。当一个作业提交时，在容器中指定了队列，那么该作业就提交到对应的队列中，也可根据配置文件中基于用户的资源要求分

配队列。每个队列中的运行作业共享该队列的资源。目前仅支持对内存的资源的公平调度，通过配置队列选项，可实现 FIFO、主要资源公平调度的方式，队列还可通过分级或配置队列权重进行资源分配。

公平调度器对每个队列有最小资源担保，这是为了确保作业、用户、群组能得到足够的资源。当一个队列中有运行状态的作业时，公平调度器保证该队列获得最小资源，当队列不需要最小资源运行作业时，队列中的空闲资源将会被其他队列划分使用。这个调度策略使得在队列中没有作业时，资源可供其他队列使用，使得资源利用率更加高效。

默认情况下，公平调度器调度所有的作业，也可以通过配置文件限制每个用户和队列运行作业的数量。这种调度在一个用户需要提交数以百计作业，或者在很多作业运行中需要产生大量中间数据或进行数据切换时，都比较有用。限制新作业的提交不会引起后面提交作业的失败，只需等待队列中用户的早期任务完成后，就可运行。

4.5.2　分级队列

公平调度器也支持分级队列，所有的队列都从 root 队列衍生而来。经典的公平调度方式是将资源分布到 root 队列的子队列中，并将分配到的资源放到子队列中，作业可在子队列中进行调度。通过配置子元素的方式指定某队列为其他队列的子队列。队列名称以父类队列的名称开始，用句点作为分隔符。

如 root 队列下的一个队列 queue1 的名称是 root.queue1，那么 parent1 下的子队列名称为 queue2，则引用方式为 root.parent1.queue2。队列中的 root 是可选项，如 queue1 可以只写 queue1，上面的 queue 可以只写 parent1.queue2。

另外，公平调度器允许用户为每个队列设置不同的用户规则来分享队列资源。用户规则通过 org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.SchedulingPolicy 进行设置。主要规则有：FIFO 规则、公平分享规则(默认)、主要资源公平规则。

公平调度器支持管理员配置策略，使提交的作业自动地放到对应队列中。公平调度器根据作业、用户、群组指定的队列信息放到指定的队列中。每个提交的作业根据策略中的规则放进一个队列。使用公平调度器，需要在 yarn-site.xml 文件中配置公平调度的类。

yarn-site.xml 中配置信息如下：

```
<property>
    <name>yarn.resourcemanager.scheduler.class</name>
    <value>
        org.apache.hadoop.yarn.server.resourcemanager
          .scheduler.fair.FairScheduler
    </value>
</property>
```

4.5.3　公平调度器队列的设置

yarn-site.xml 中，公平调度器的配置信息如下：

```
<?xml version="1.0"?>
<allocations>
    <queue name="sample_queue">
        <minResources>10000 mb,0vcores</minResources>
```

```
        <maxResources>90000 mb,0vcores</maxResources>
        <maxRunningApps>50</maxRunningApps>
        <maxAMShare>0.1</maxAMShare>
        <weight>2.0</weight>
        <schedulingPolicy>fair</schedulingPolicy>
        <queue name="sample_sub_queue">
            <aclSubmitApps>charlie</aclSubmitApps>
            <minResources>5000 mb,0vcores</minResources>
        </queue>
    </queue>

    <queueMaxAMShareDefault>0.5</queueMaxAMShareDefault>

    <!-- Queue 'secondary_group_queue' is a parent queue and may have
      user queues under it -->
    <queue name="secondary_group_queue" type="parent">
        <weight>3.0</weight>
    </queue>

    <user name="sample_user">
        <maxRunningApps>30</maxRunningApps>
    </user>
    <userMaxAppsDefault>5</userMaxAppsDefault>
    <queuePlacementPolicy>
        <rule name="specified" />
        <rule name="primaryGroup" create="false" />
        <rule name="nestedUserQueue">
            <rule name="secondaryGroupExistingQueue" create="false" />
        </rule>
        <rule name="default" queue="sample_queue"/>
    </queuePlacementPolicy>
</allocations>
```

配置格式文件必须是 XML 文件，包含多种类型的元素。

(1) 队列元素：队列元素可以带一个选项属性 type，可以设置为 parent，使它成为一个父队列。

在需要配置一个父队列而不需要子队列的场合，每个队列元素可以包含下面的属性。

① minResources：通过“X mb, Y vcores”格式，设置队列的最小资源数。对于单类型资源(内存)的公平策略，vcores 的值是被忽略的。当一个队列的资源不能满足时，公平调度器将根据单资源公平策略重新分配。一个队列的资源是否被满足，主要看其获得的内存是否达到了最小资源要求。如果多个队列出现不满足最小资源需求的情况，那么，资源将按照使用资源与最小资源的比值最小化来进行分配，即倾向于按每个队列的最小资源配额进行资源分配。一个队列有新提交作业时，该作业可能不能立即获得最小资源，因为已运行的作业正占用着队列的资源。

② maxResources：通过“X mb, Y vcores”格式，设置队列的最大资源数。对于单类型资源(内存)的公平策略，vcores 的值是被忽略的。队列获得的资源的总和不能超过该最大资源限制数。

③ maxRunningApps：限制队列同时运行的作业数。

④ maxAMShare：设置可用于运行 Application Master 的资源比例。该属性可只在子队列中使用。例如设置为 1.0f，则表示 Application Master 可以占用 100%的内存和 CPU 资源，而当值为-1.0f 时，表示跳过该属性检查。默认情况下，该值为 0.5f。

⑤ Weight：设置非等比的占有集群资源的比重。Weight 默认值为 1，当权重为 2 时，表示该队列将得到大约 2 倍于默认配置的资源。

⑥ schedulingPolicy：设置队列的调度规则。该属性的可选项为：fifo、fair、drf 或自研调度规则类型。但必须继承并实现 org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.SchedulingPolicy 的类及方法。该属性的默认值为 fair。如果选择 fifo，表示早提交的作业会优先获得足够的容器，后提交的作业在剩余资源能满足需求的情况下才可以同时执行，不满足时则等待资源。

⑦ aclSubmitApps：设置可向该队列提交作业的用户或群组。

⑧ aclAdministerApps：设置可管理队列的用户或群组列。

⑨ minSharePreemptionTimeout：设置队列最小配额不满足的等待时长。超时后，该队列将优先从其他队列中获得资源。若不设置该值，则将继承父队列的值。

⑩ fairSharePreemptionTimeout：设置队列平均配额不满足的等待时长。超时后，该队列将优先从其他队列中获得资源。若不设置该值，则将继承父队列的值。

(2) 用户元素：设置每个用户的属性。最小化配置时，仅需要设置一个属性，即 maxRunningApps，该属性表示该用户可同时运行作业数的最大值。

(3) 默认用户最大作业数元素：设置用户默认的最大同时运行作业的数量。对于那些没有指定同时运行任务数限制的用户，采用该默认设置。

(4) 默认获取公平资源超时元素：设置 root 队列对公平资源的最大等待时长，可通过 root 队列的 fairSharePreemptionTimeout 进行覆盖设置。

(5) 默认获取最小资源超时元素：设置 root 队列对最小资源的最大等待时长，可通过 root 队列的 minSharePreemptionTimeout 进行覆盖设置。

(6) 默认获取公平资源阈值元素：设置 root 队列对公平资源的分配阈值，可通过 root 队列的 fairSharePreemptionThreshold 进行覆盖设置。

(7) 默认队列最大作业数元素：设置队列默认可以运行的最大作业数，可通过队列的 maxRunningApps 进行覆盖设置.

(8) 默认 AM 对队列资源的使用限制元素：设置 Application Master 对队列资源的使用限制。可通过 maxAMShare 进行覆盖设置。

(9) 默认队列调度规则元素：设置队列的默认调度规则。队列可以通过设置 scheduling Policy 进行覆盖重载，默认调度规则为 fair。

(10) 作业放置规则元素：设置调度器如何将提交的作业放置到队列中的规则，规则的使用顺序严格按照配置顺序。规则可以带参数，如所有规则都可传 create 参数，create 默认为 true，表示这个规则可创建新的队列。若设置为 false，则表示把作业放置到不可创建文件的队列中。最后的规则必须有终止符，合法的规则如下。

① specified：设置作业放置的它队列。如果一个作业没有指定队列，则选择默认 default 队列。

② user：用户提交的作业放进该用户对应的队列中。

③ primaryGroup：用户提交的作业放进用户所在群组对应的队列中。

④ secondaryGroupExistingQueue：用户提交的作业放置到设置的第二群组所对应的队列中。

⑤ nestedUserQueue：用户提交的作业放进嵌套规则的队列中。这与 user 规则有点相似，不同的是，nestedUserQueue 可以在父队列下创建队列，而 user 只有在 root 队列下才可以创建新队列。nestedUserQueue 规则被使用的条件是嵌套规则返回一个父队列，用户可以直接配置父队列，或通过设置队列的 type 属性，指定为 parent。

⑥ default：用户提交的作业放置到默认规则中配置的 queue。如果 queue 没有配置，作业将被放到 root.default 队列。

⑦ reject：拒绝用户提交作业。

4.6 资源管理者(RM)重启机制

资源管理者(Resource Manager)是 YARN 系统中负责管理资源和调度应用的核心模块。

4.6.1 什么是资源管理器重启

我们知道，在 Hadoop2 之前的版本中，Jobtracker 是单点，在 Hadoop2 中，资源管理器 RM 同样是集群的一个单点。但 RM 通过重启资源管理器技术，实现了资源管理器的宕机时间对于终端用户来说是不可见、不易察觉的。

资源管理器(RM)重启机制分为两大类。

(1) 资源管理器重启 1(非工作保存 RM 重启)。

为增强资源管理器持续执行作业的能力，RM 将持久化保存作业的元数据信息到状态存储器中。RM 重启后，将从状态存储器中重载这些信息，并重启先前运行的作业，且用户不需要重复提交作业。运行中的作业会被 kill 掉，重启后重新执行。

(2) 资源管理器重启 2(工作保存 RM 重启)。

通过合并节点管理器上的容器状态信息和重启时应用管理者中的容器请求重建资源管理器的运行状态。与非工作保存 RM 重启方式的关键不同点，是先前运行的作业在 RM 重启后不会被 kill，作业不会因为 RM 的丢失而影响执行。

4.6.2 非工作保存 RM 重启

资源管理器重启 1(非工作保存 RM 重启)的核心思想是：把用户提交的作业信息以及作业的最后状态信息、作业结束时的诊断信息都持久化到一个可插的存储中，另外，RM 也保存认证信息，如安全环境中的 key、tokens。当 RM 宕机重启后，RM 从可从存储中请求作业的信息(元数据信息、认证环境中的认证信息)，根据这些信息，RM 将重建作业，并重新提交。如果作业的状态是 finished 或 failed 或 killed，RM 重启后就不会重新提交。

节点管理器(NN)和客户端在 RM 宕机期间保持着先前的 RM 状态，直到新的 RM 启动起来。当新的 RM 启动完成后，RM 将通过心跳发送同步命令给所有的 NodoManager 和 ApplicationMasters。

NodeManagers 和 ApplicationMasters 收到重新同步命令后的处理逻辑如下。

(1) NM 会 kill 所有它所管理的容器并重新向 RM 注册，RM 的内部机制会把重新注册的 NodeManager 当作新节点来处理。

(2)　AM(如 MapReduce AM)收到重新同步命令后将会退出。

RM 重启后会从状态存储中加载所有的作业元信息，包括获取作业提交信息、状态信息、认证信息，并填载到内存中，RM 会为每一个未完成的作业创建一个新的 attempt(例如 MapReduce 的 ApplicationMaster)。在这种模式下，先前运行的作业会丢失，因为这些作业在 RM 重启后的重新同步命令中全部被退出。

4.6.3　工作保存 RM 重启

Hadoop 2.6.0 版本中，RM 重启的特性得到了极大的发展，实现了在 RM 重启后作业的继续执行而不会被强制退出并重新提交。

在第一个方案中，保存了作业提交信息、认证信息，以及恢复所需要的状态信息，根据这些信息重建该作业。第二个方案的关键点不是重建可提交作业，而是重建执行中的作业。难点是要实现对 RM 中所有生命周期内容器的状态跟踪、作业的 headroom、资源的请求、队列资源利用率信息的收集。这种 RM 重启方式中，RM 不需要 kill AM，正在执行的作业也不会强制终止，而是继续执行。真正实现了 RM 重启对用户、提交的作业都是不可见、不易发现的。最重要的是，工作保留 RM 重启方式能节省大量的作业重跑时间，尤其是那些需要运行多天的大作业。

RM 恢复作业的运行状态是通过从 NMs 发送来的容器状态获取的。每个 NM 在接收到重新同步命令后，不会 kill 容器，而是直接进行节点注册，在容器再注册后，RM 将继续管理容器并发送容器状态，并重建容器实例和作业的调度状态。AM 需要重新发送未满足的资源请求的作业给 RM，因为 RM 在宕机后会丢失未满足的资源请求的作业信息。

4.6.4　RM 重启配置 yarn-site.xml

重启配置文件如下所示：

```
<property>
   <description>Enable RM to recover state after starting. If true, then
     yarn.resourcemanager.store.class must be specified</description>
   <name>yarn.resourcemanager.recovery.enabled</name>
   <value>true</value>
</property>
<property>
   <description>The class to use as the persistent store.</description>
   <name>yarn.resourcemanager.store.class</name>
   <value>org.apache.hadoop.yarn.server.resourcemanager.recovery
     .ZKRMStateStore</value>
</property>
<property>
   <description>Comma separated list of Host:Port pairs. Each corresponds
     to a ZooKeeper server
     (e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002") to be used by the
     RM for storing RM state.
     This must be supplied when using
     org.apache.hadoop.yarn.server.resourcemanager.recovery
     .ZKRMStateStore
     as the value for yarn.resourcemanager.store.class</description>
   <name>yarn.resourcemanager.zk-address</name>
   <value>127.0.0.1:2181</value>
</property>
```

4.7　资源管理器的高可用性(RM HA)

高可用机制可避免单点问题，满足生产环境中的资源实时可用要求。

4.7.1　什么是资源管理器的高可用性

这里介绍 YARN RM 的高可用性以及如何配置。RM 负责集群资管理和作业调度。在 Hadoop 2.4 之前，RM 是 YARN 集群的单点，因为还没有加入 RM 重启机制，高可用部署是一个非常值得做的工作。RM 高可用的实现思想与传统的思想是一致的，就是要实现 RM 的主/备冗余结构，以去掉单点问题。如图 4-3 所示。

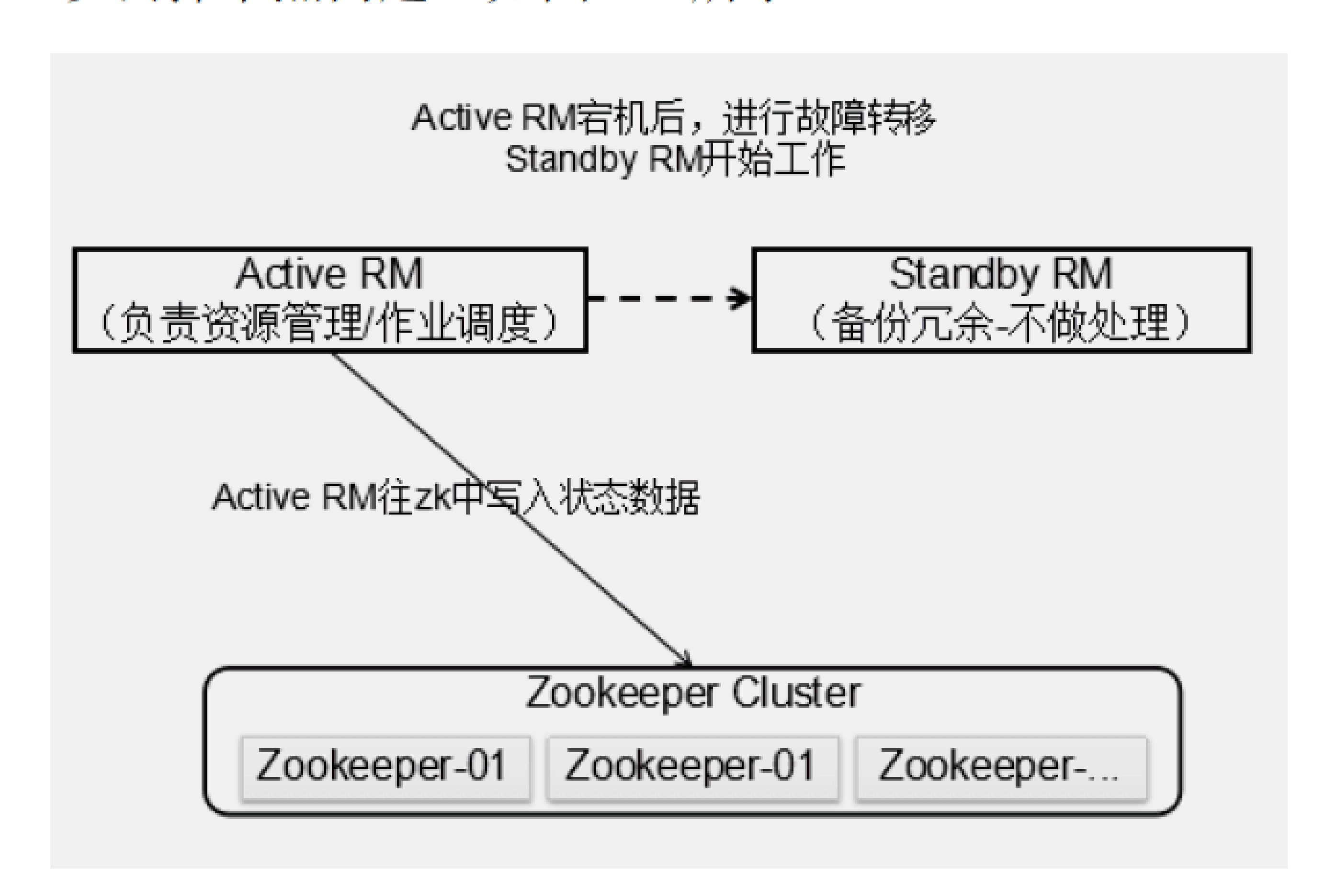


图 4-3　资源管理器的高可用性架构

RM HA 通过主/备架构来实现，在任一时刻，有且仅有一个 RM 是活跃的，另外的一个或多个处于备份模式，并等待接管为主机的角色。从备机到主机的触发条件可以通过管理员命令完成，也可以通过 YARN 中配置的失败转移控件自动实现。

4.7.2　自动故障转移

资源管理器(RM)可通过设置 Zookeeper-based ActiveStandbyElector 来决定哪个 RM 作为活跃 RM。当活跃 RM 宕机或者由于网络原因不应答，备份 RM 会自动被选举为活跃 RM，并接管 RM 的所有工作。这种方式可及时解决故障导致的服务停滞，比通过管理员手动指定新的 Active RM 要高效。当然，管理员可以指定硬件配置已有的 Standby RM 作为新的 Active RM，这是自动故障转移机制所不具有的。

4.7.3　客户端/应用管理器/节点管理器的故障转移

当有多个 RM 时，需要在 yarn-site.xml 中列出所有的 RM。客户端、应用管理器、节点管理器以循环赛的形式选择并连接 RM，直到它们命中活跃的 RM。如果活跃节点宕了，某个 Standby RM 将成为新的 Active RM，客户端、应用管理器、节点管理器将再次以循环赛

的形式选择并连接RM，直到命中这个新的RM。默认的选择连接逻辑是在 org.apache.hadoop.yarn.client.ConfiguredRMFailoverProxyProvider 类中实现的。可以通过重写 org.apache.hadoop.yarn.client.RMFailoverProxyProvider 实现自有的策略。当我们指定自有策略时，需要设置 yarn.client.failover-proxy-provider 的值为该类的类名，内部通过反射机制实例化该类对象并执行对应的逻辑。

4.7.4　部署 RM HA

故障转移功能是通过多个配置项协调完成的，下面是一些必需的或重要的配置属性，如表 4-2 所示。在 yarn-default.xml 中有全部的配置属性，读者可以自行查看。

表 4-2　RM HA 的相关属性字段及配置说明

配置属性	属性说明
yarn.resourcemanager.zk-address	Zookeeper 集群的地址，为状态存储器、嵌入式选举器提供服务
yarn.resourcemanager.ha.enabled	打开 RM HA 功能
yarn.resourcemanager.ha.rm-ids	RM id，例如 rm1、rm2
yarn.resourcemanager.hostname.rm-id	为每一个 rm-id 指定 hostname，可设置为每个 RM 的服务地址
yarn.resourcemanager.address.rm-id	为每个 rm-id 指定地址(host:port)，方便用户提交作业。若设置了该属性，则将会覆盖 yarn.resourcemanager.hostname.rm-id 中设置的 rm-id
yarn.resourcemanager.scheduler.address.rm-id	为每个 rm-id 指定调度器地址(host:id)。若设置了该属性，则覆盖在 yarn.resourcemanager.hostname.rm-id 中设置的 hostname
yarn.resourcemanager.resource-tracker.address.rm-id	为每个 rm-id 指定连接 NM 的地址(host:port)。若设置了该属性，则将覆盖在 yarn.resourcemanager.hostname.rm-id 中设置的 hostname
yarn.resourcemanager.admin.address.rm-id	为每个 rm-id 指定管理地址(host:port)。若设置该了属性，则将覆盖 yarn.resourcemanager.hostname.rm-id 中设置的 hostname
yarn.resourcemanager.webapp.address.rm-id	为每个 rm-id 指定 Web 应用地址(host:port)。若设置了 yarn.http.policy 为 HTTPS_ONLY，该属性就不必再设置。该设置将会覆盖在 yarn.resourcemanager.hostname.rm-id 中设置的 hostname
yarn.resourcemanager.webapp.https.address.rm-id	为 rm-id 指定 HTTPS 的地址(host:port)。若设置了 yarn.http.policy 为 HTTP_ONLY，则不需要在设置该属性。该属性会覆盖 yarn.resourcemanager.hostname.rm-id 中设置的 hostname

续表

配置属性	属性说明
yarn.resourcemanager.ha.id	RM 在高可用集群中的 id。如果设置该属性，管理员必须确定每个 RM 在配置文件中都拥有唯一的 ID
yarn.resourcemanager.ha.automatic-failover.enabled	打开自动故障转移功能，默认情况下，自动故障转移是打开状态
yarn.resourcemanager.ha.automatic-failover.embedded	在自动故障转移打开时，使用嵌入式选举法来得到新的活跃 RM。默认情况下，该功能在 HA 可用时为打开状态
yarn.resourcemanager.cluster-id	集群的 id
yarn.client.failover-proxy-provider	客户端、应用管理器、节点管理器故障转移到活跃 RM 的处理类，可用自带类，也可用自研类
yarn.client.failover-max-attempts	FailoverProxyProvider 进行故障转移的最大尝试次数
yarn.client.failover-sleep-base-ms	设置故障转移休眠底数，单位是毫秒，用于计算故障转移之间的幂指数延迟
yarn.client.failover-sleep-max-ms	设置故障转移时的最大时长
yarn.client.failover-retries	设置故障转移时尝试连接 RM 的最大次数
yarn.client.failover-retries-on-socket-timeouts	设置连接 RM 的最大超时

4.7.5 配置例子

在 yarn-site.xml 中配置如下：

```
<property>
    <name>yarn.resourcemanager.ha.enabled</name>
    <value>true</value>
</property>
<property>
    <name>yarn.resourcemanager.cluster-id</name>
    <value>cluster1</value>
</property>
<property>
    <name>yarn.resourcemanager.ha.rm-ids</name>
    <value>rm1,rm2</value>
</property>
<property>
    <name>yarn.resourcemanager.hostname.rm1</name>
    <value>master1</value>
</property>
<property>
    <name>yarn.resourcemanager.hostname.rm2</name>
    <value>master2</value>
</property>
<property>
    <name>yarn.resourcemanager.webapp.address.rm1</name>
    <value>master1:8088</value>
</property>
<property>
    <name>yarn.resourcemanager.webapp.address.rm2</name>
```

```
    <value>master2:8088</value>
</property>
<property>
    <name>yarn.resourcemanager.zk-address</name>
    <value>zk1:2181,zk2:2181,zk3:2181</value>
</property>
```

4.7.6　管理员命令

YARN 的 RM 管理员(启动 YARN 的用户)有很多 HA 命令，用以检查 RM 的健康/状态、转变主/备角色。HA 以 id 作为参数(在参数表中有介绍)选择活跃 RM 的命令，该参数在 yarn.resourcemanager.ha.rm-ids 中设置。例如，查看指定 rm 的角色：

```
$ yarn rmadmin -getServiceState rm1
active
$ yarn rmadmin -getServiceState rm2
standby
```

在自动故障转移打开的情况下，管理员不需要手动进行角色切换，并且 RM 会拒绝人工切换。如果非常明确需要进行人工干预切换，可通过加上-forcemanual 强制进行手动切换。

(1)　在没有开启自动故障转移的情况下，手动进行切换：

```
$ yarn rmadmin -transitionToStandby  rm1
```

(2)　在开启自动故障转移的情况下，手动进行强制切换：

```
$ yarn rmadmin -transitionToStandby -forcemanual rm1
```

4.8　节 点 标 签

节点标签是用相似的字符来组织节点的方法，用户提交作业可以指定标签。

4.8.1　节点标签的特点

节点标签的特点如下。

(1)　一个节点只能有一个节点分区，因此，一个集群可以通过分区变为几个不相关的子集群。默认情况下，节点的默认标签是 partition=“”。

(2)　用户需要配置每个分区有多少资源可以被不同的队列使用。

(3)　用户可指定节点标签集，这些节点可以被每个队列访问，单个应用通过所在的队列来使用节点标签集中的一个子集。

4.8.2　节点标签的属性

节点标签的属性如下。

(1)　分区集群：每个节点可以分配一个标签，所以一个集群可通过设置不同的标签，划分成多个小的、不相关的分区集群。

(2)　设置队列对节点标签的 ACL：用户可以设置每个队列可以访问的节点标签集，很

多节点只能被指定的队列访问。这与队列申请资源不同，资源申请得到的资源可以是集群的中任意一节点，而指定了节点标签集的队列，只能从指定的节点集中获取资源。

(3) 指定队列对分区资源的使用百分比：用户可以设置队列对指定标签节点资源使用的百分比。如队列 A 可以访问标签为 hbase 的节点上的 30%的资源，注意该百分比设置要与已经存在的资源管理分配一致。

4.8.3 节点标签的配置

在 RM 中配置节点标签，在 yarn-site.xml 中的配置如表 4-3 所示。

表 4-3 节点标签的配置属性

配置属性	属 性 值
yarn.node-labels.fs-store.root-dir	hdfs://namenode:port/path/to/store/node-labels/
yarn.node-labels.enabled	true

注意：确保 yarn.node-labels.fs-store.root-dir 配置的目录被创建并且 RM 启动账号有权限访问该目录，并有读、写权限。如果用户想用本地化的文件系统来存储节点标签信息，以代替 HDFS，则路径格式为：file:///home/yarn/node-label。集群管理员可通过命令增加/修改节点标签列表、节点标签映射到 YARN 中。增加节点标签列表：

```
$ yarn rmadmin -addToClusterNodeLabels
"label_1(exclusive=true/false),label_2(exclusive=true/false)"
//增加节点标签。如果用户不指定“(exclusive=…)”，则 exclusive=true 是默认值
$ yarn cluster --list-node-labels //查看新添加的节点标签是否加到了集群中
```

给节点增加标签：

```
$ yarn rmadmin -replaceLabelsOnNode "node1[:port]=label1 node2=label2"
//node1 增加标签 label1，node2 增加标签 label2
```

4.8.4 使用节点标签的调度器配置

表 4-4 列出了节点标签的调度配置属性。

表 4-4 节点标签的调度配置属性

配置属性	属性描述
yarn.scheduler.capacity.<queue-path>.capacity	设置队列访问节点中默认分区资源的百分比。队列的子队列对默认分区使用量的总和必须等于 100
yarn.scheduler.capacity.<queue-path>.accessible-node-labels	设置管理员需要指定每个队列可以访问的节点标签。用逗号隔开，如“hbase,storm”，表示该队列可以访问节点标签 hbase 和 storm。所有的队列可以访问没有标签化的节点，用户也不需要指定标签。如果用户不设置该属性，该值将要从它的父队列中继承该值。如果用户要明确地指定队列可以访问所有未标签化的节点，只需要将该属性的值设置为空即可

续表

配置属性	属性描述
yarn.scheduler.capacity.<queue-path>.accessible-node-labels.<label>.capacity	设置队列访问指定标签<label>的资源百分比。子队列对标签<label>的容量的总和必须为 100，默认该属性的值为 0
yarn.scheduler.capacity.<queue-path>.accessible-node-labels.<label>.maximum-capacity	设置队列可以访问指定节点标签的最大容量，该属性的默认值是 100
yarn.scheduler.capacity.<queue-path>.default-node-label-expression	设置当用户提交任务而又没有指定节点标签时的默认节点标签。该属性默认为空

4.8.5 节点标签配置示例

假设队列结构为：

```
              root
           /    |    \
    engineer   sales  marketing
```

集群中有 5 个节点(hostname=h1..h5)，每个节点有 14GB 的内存，24 核。其中一个有 GPU(假设是 h5)，所以管理员增加 GPU 标签给 h5。

容量调度器的配置如下：

```
yarn.scheduler.capacity.root.queues=engineering,marketing,sales
yarn.scheduler.capacity.root.engineering.capacity=33
yarn.scheduler.capacity.root.marketing.capacity=34
yarn.scheduler.capacity.root.sales.capacity=33

yarn.scheduler.capacity.root.engineering.accessible-node-labels=GPU
yarn.scheduler.capacity.root.marketing.accessible-node-labels=GPU

yarn.scheduler.capacity.root.engineering.accessible-node-labels.GPU.capacity=50
yarn.scheduler.capacity.root.marketing.accessible-node-labels.GPU.capacity=50

yarn.scheduler.capacity.root.engineering.default-node-label-expression=GPU
```

如上面 root.engineering/marketing/sales.capacity=33 表示 3 个子队列可以获得非分区资源的 1/3。所以，它们可以使用 h1..h4 的 1/3 的资源，等于 24*4*(1/3)=(32GB 内存，32 核)。只有 engineering/marketing 队列有权限可以访问 GUP 分区。engineering/marketing 队列平占该 GPU 分区的资源。它们可以使用 h5 资源的 1/2，值为 24*0.5=(12GB 内存，12 核)。

注意如下两点。

(1) 在完成配置 CapacityScheduler 之后，需要执行 yarn rmadmin -refreshQueues，来加载修改。

(2) 到 RM 的 WebUI 界面检查节点标签配置。

4.8.6 指定应用的节点标签

提交作业时，可用下面的 Java API 指定节点标签以请求资源。

(1) 作业可以通过 ApplicationSubmissionContext.setNodeLabelExpression(...)，来设置节点标签。

(2) 通过方法 ResourceRequest.setNodeLabelExpression(...)为每个资源请求设置节点标签，该方法可以被 ApplicationSubmissionContext 方法覆盖。

(3) 通过 ApplicationSubmissionContext 中的 setAMContainerResourceRequest.setNodeLabelExpression 指定作业 master 容器的节点标签。

4.8.7 节点标签的监控

(1) 通过 Web UI，可以看到如下信息。

① 通过 http://RM-Address:port/cluster/nodes，查看每个节点上的标签。

② 通过 http://RM-Address:port/cluster/nodelabels，可以查看活跃节点的数量，每个分区的总资源量。

③ 通过 http://RM-Address:port/cluster/scheduler，可以查看每个队列设置的相关节点标签，以及对这些资源的利用率。

(2) 通过命令行监控。

用 yarn cluster --list-node-labels 来获得集群中的节点标签：

```
$ yarn cluster --list-node-labels
```

用 yarn node -status <NodeId>来获取指定的节点标签的状态：

```
$ yarn node -status <NodeId>
```

4.9 YARN 编程

该部分介绍如何在 YARN 上实现新应用的方法。目前，默认支持的应用类型有 MapReduce、Spark、Storm。组织或用户可以开发适合自己的处理框架，当然，这是一件很有挑战性的工作。

4.9.1 什么是 YARN 级别编程

当一个作业通过客户端提交到 YARN 的资源管理器(RM)时，RM 将根据资源请求分配容器启动 AM，由 NM 与 AM 共同完成对该作业的启动和管理。作业提交客户端可以通过创建 YarnClient 对象来实现。在 YarnClient 启动后，客户端可以创建作业上下文，为作业申请第一容器，该容器包含 ApplicationMaster(AM)，并提交任务到 AM。用户需要给出必需的作业信息，包括作业运行的 JAR 包、执行命令、参数、环境设置等。

AM 在 YARN RM 上申请的第一容器中启动。该 AM 与 YARN 建立通信，并处理作业的执行，AM 以异步模式进行工作。在启动作业的过程中，AM 的主要任务如下。

(1) 与 RM 通信并协商、分配作业容器的资源，这些容器资源在 NM 上开辟。

(2) 在容器分配后，与 NM 通信，在 NM 上启动应用容器。由此可以认为，启动作业的是 AM，任务(task)的运行容器在 NM 上。Task a 通过 AMRMClientAsync 以异步模式完成，

需要在 AMRMClientAsync.CallbackHandler 指定事件处理方法。Task b 在容器分配后，通过可执行对象来启动容器。AM 必须为这个容器指定 ContainerLaunchContext，该对象中包含启动信息，如命令行中指定的信息、环境等。

在作业执行过程中，AM 通过 NMClientAsync 对象与 NMs 进行通信。所有的容器事件是通过 NMClientAsync.CallbackHandler 来处理的。常用的回调操作有 start、stop、status update 和 error。AM 通过 AMRMClientAsync.CallbackHandler 的方法 getProgress()向 AM 报告执行进度等信息。

4.9.2 YARN 的相关接口

下面是写 Application Master 的重要接口，主要包括作业客户端与 RM、AM 与 RM 和 NM 交互接口。

(1) Client<-->ResourceManager：通过使用 YarnClient 对象完成，实现客户端与资源管理器(RM)的通信。

(2) ApplicationMaster<-->ResourceManager：通过 AMRMClientAsync.CallbackHandler 异步处理事件。

(3) ApplicationMaster<-->NodeManager：启动容器，AM 通过 NMClientAsync 对象与 NM 进行通信，通过 NMClientAsync.CallbackHandler 处理容器事件。

注意：

- YARN 应用的三个主要协议是 ApplicationClientProtocol、ApplicationMasterProtocol 和 ContainerManagementProtocol。这三个客户端打包了三个协议，以简单的程序模块为 YARN 应用服务。
- 编程人员可以直接使用这三个协议实现应用，但不提倡用户这样使用，而是推荐基于三个客户端进行新应用管理器的开发。

4.9.3 编程实践

1. 编写一个新应用类型的客户端

(1) 初始化 YarnClient：

```
YarnClient yarnClient = YarnClient.createYarnClient();
yarnClient.init(conf);
yarnClient.start();
```

其中 createYarnClient 为创建 YarnClient 的构造函数，无参数。init 对客户端进行初始化，start 启动客户端。

YarnClient.CreateYarnClient()源码为：

```
/**
 * Create a new instance of YarnClient.
 */
@Public
public static YarnClient createYarnClient() {
   YarnClient client = new YarnClientImpl();
   return client;
}
```

(2) 初始化 YarnClientApplication：

```
YarnClientApplication app = yarnClient.createApplication();
GetNewApplicationResponse appResponse = app.getNewApplicationResponse();
```

在一个新应用创建后的返回值中，包含了集群的信息，比如集群的最小/最大资源容量。这些信息非常重要，只有如此，才能保证用户可以获得指定的容器，以启动、运行 AM。

yarnClient.createApplication()的函数定义：

```
/**
 * Obtain a {@link YarnClientApplication} for a new application,
 * which in turn contains the {@link ApplicationSubmissionContext} and
 * {@link  org.apache.hadoop.yarn.api.protocolrecords
 *  .GetNewApplicationResponse}
 * @return {@link YarnClientApplication} built for a new application
 */
public abstract YarnClientApplication createApplication()
  throws YarnException, IOException;
```

(3) 客户端的主要职责是设置 RM 启动 AM 所需要的所有信息，客户端通过 ApplicationSubmissionContext 来设置如下信息：

① 作业信息：id、名称。

② 队列、优先级：作业要提交到的队列，应用执行的优先级。

③ 用户：提交作业的用户。

④ 容器启动上下文(ContainerLaunchContext)：启动 AM 的容器的信息，ContainerLaunchContext 定义了所有运行应用需要的信息，例如本地资源信息(执行文件、JAR 包、文件等)，环境设置(类路径等)等，执行的命令和验证口令。

```
//设置应用的提交上下文
//创建上下文对象
ApplicationSubmissionContext appContext =
  app.getApplicationSubmissionContext();
//从上下文中获取应用 id
ApplicationId appId = appContext.getApplicationId();

//设置 keepContainers，设置应用名
appContext.setKeepContainersAcrossApplicationAttempts(keepContainers);
appContext.setApplicationName(appName);

//设置 AM 需要的本地资源
//AM 需要的本地文件或文档
//AM 需要的 JAR 包，也是本地资源的组成部分
Map<String, LocalResource> localResources =
  new HashMap<String, LocalResource>();

LOG.info(
 "Copy App Master jar from local filesystem and add to local environment");
//复制 AM 需要的 JAR 包到文件系统
//创建一个本地资源对象，指向目的 JAR 路径
FileSystem fs = FileSystem.get(conf);
addToLocalResources(fs, appMasterJar, appMasterJarPath, appId.toString(),
  localResources, null);

//Set the log4j properties if needed
if (!log4jPropFile.isEmpty()) {
    addToLocalResources(fs, log4jPropFile, log4jPath, appId.toString(),
```

```
      localResources, null);
}

//The shell script has to be made available on the final container(s)
//where it will be executed.
//To do this, we need to first copy into the filesystem that is visible
//to the yarn framework.
//We do not need to set this as a local resource for the application
//master as the application master does not need it.
String hdfsShellScriptLocation = "";
long hdfsShellScriptLen = 0;
long hdfsShellScriptTimestamp = 0;
if (!shellScriptPath.isEmpty()) {
    Path shellSrc = new Path(shellScriptPath);
    String shellPathSuffix =
      appName + "/" + appId.toString() + "/" + SCRIPT_PATH;
    Path shellDst =
      new Path(fs.getHomeDirectory(), shellPathSuffix);
    fs.copyFromLocalFile(false, true, shellSrc, shellDst);
    hdfsShellScriptLocation = shellDst.toUri().toString();
    FileStatus shellFileStatus = fs.getFileStatus(shellDst);
    hdfsShellScriptLen = shellFileStatus.getLen();
    hdfsShellScriptTimestamp = shellFileStatus.getModificationTime();
}
if (!shellCommand.isEmpty()) {
    addToLocalResources(fs, null, shellCommandPath, appId.toString(),
      localResources, shellCommand);
}
if (shellArgs.length > 0) {
    addToLocalResources(fs, null, shellArgsPath, appId.toString(),
    localResources, StringUtils.join(shellArgs, " "));
}
//Set the env variables to be setup in the env where the application master
//will be run
LOG.info("Set the environment for the application master");
Map<String, String> env = new HashMap<String, String>();
//put location of shell script into env
//using the env info, the application master will create the correct local
//resource for the
//eventual containers that will be launched to execute the shell scripts
env.put(DSConstants.DISTRIBUTEDSHELLSCRIPTLOCATION,
  hdfsShellScriptLocation);
env.put(DSConstants.DISTRIBUTEDSHELLSCRIPTTIMESTAMP,
  Long.toString(hdfsShellScriptTimestamp));
env.put(DSConstants.DISTRIBUTEDSHELLSCRIPTLEN,
  Long.toString(hdfsShellScriptLen));
//Add AppMaster.jar location to classpath
//At some point we should not be required to add
//the hadoop specific classpaths to the env.
//It should be provided out of the box.
//For now setting all required classpaths including
//the classpath to "." for the application jar
StringBuilder classPathEnv = new StringBuilder(Environment.CLASSPATH.$$())
  .append(ApplicationConstants.CLASS_PATH_SEPARATOR).append("./*");
for (String c : conf.getStrings(
  YarnConfiguration.YARN_APPLICATION_CLASSPATH,
  YarnConfiguration.DEFAULT_YARN_CROSS_PLATFORM_APPLICATION_CLASSPATH)) {
    classPathEnv.append(ApplicationConstants.CLASS_PATH_SEPARATOR);
    classPathEnv.append(c.trim());
}
classPathEnv.append(ApplicationConstants.CLASS_PATH_SEPARATOR)
```

```
  .append("./log4j.properties");
//Set the necessary command to execute the application master
Vector<CharSequence> vargs = new Vector<CharSequence>(30);
//Set java executable command
LOG.info("Setting up app master command");
vargs.add(Environment.JAVA_HOME.$$() + "/bin/java");
//Set Xmx based on am memory size
vargs.add("-Xmx" + amMemory + "m");
//Set class name
vargs.add(appMasterMainClass);
//Set params for Application Master
vargs.add("--container_memory " + String.valueOf(containerMemory));
vargs.add("--container_vcores " + String.valueOf(containerVirtualCores));
vargs.add("--num_containers " + String.valueOf(numContainers));
vargs.add("--priority " + String.valueOf(shellCmdPriority));
for (Map.Entry<String, String> entry : shellEnv.entrySet()) {
    vargs.add("--shell_env " + entry.getKey() + "=" + entry.getValue());
}
if (debugFlag) {
    vargs.add("--debug");
}
vargs.add("1>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR
  + "/AppMaster.stdout");
vargs.add("2>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR
  + "/AppMaster.stderr");
//Get final commmand
StringBuilder command = new StringBuilder();
for (CharSequence str : vargs) {
    command.append(str).append(" ");
}
LOG.info("Completed setting up app master command " + command.toString());
List<String> commands = new ArrayList<String>();
commands.add(command.toString());
//Set up the container launch context for the application master
ContainerLaunchContext amContainer = ContainerLaunchContext.newInstance(
  localResources, env, commands, null, null, null);
//Set up resource type requirements
//For now, both memory and vcores are supported, so we set memory and
//vcores requirements
Resource capability = Resource.newInstance(amMemory, amVCores);
appContext.setResource(capability);
//Service data is a binary blob that can be passed to the application
//Not needed in this scenario
//amContainer.setServiceData(serviceData);
//Setup security tokens
if (UserGroupInformation.isSecurityEnabled()) {
    //Note: Credentials class is marked as LimitedPrivate for HDFS
    //and MapReduce
    Credentials credentials = new Credentials();
    String tokenRenewer = conf.get(YarnConfiguration.RM_PRINCIPAL);
    if (tokenRenewer == null || tokenRenewer.length() == 0) {
        throw new IOException(
        "Can't get Master Kerberos principal for the RM to use as renewer");
    }
    //For now, only getting tokens for the default file-system.
    final Token<?> tokens[] =
      fs.addDelegationTokens(tokenRenewer, credentials);
    if (tokens != null) {
        for (Token<?> token : tokens) {
            LOG.info("Got dt for " + fs.getUri() + "; " + token);
        }
```

```
        }
        DataOutputBuffer dob = new DataOutputBuffer();
        credentials.writeTokenStorageToStream(dob);
        ByteBuffer fsTokens =
          ByteBuffer.wrap(dob.getData(), 0, dob.getLength());
        amContainer.setTokens(fsTokens);
    }
    appContext.setAMContainerSpec(amContainer);
```

(4)　创建过程完成后，客户端指定优先级、队列并提交作业：

```
//Set the priority for the application master
Priority pri = Priority.newInstance(amPriority);
appContext.setPriority(pri);
//Set the queue to which this application is to be submitted in the RM
appContext.setQueue(amQueue);
//Submit the application to the applications manager
//SubmitApplicationResponse submitResp =
//  applicationsManager.submitApplication(appRequest);
yarnClient.submitApplication(appContext);
```

此时，RM 会收到作业提交，并在后台根据指定资源要求分配一个 master 容器，以此启动 AM。

客户端有多种方式可以跟踪作业的进度。如客户端可以与 RM 建立连接，通过 YarnClient 的 getApplicationReport()方法获得应用的报告信息：

```
//Get application report for the appId we are interested in
ApplicationReport report = yarnClient.getApplicationReport(appId);
```

从 RM 获得的应用报告包含下面的信息。

①　通用作业信息：作业 id、队列、提交用户和启动时间。

②　AM 详细信息：AM 运行的 host、rpc 端口。AM 侦听该端口并及时响应来自客户端的请求。

③　应用跟踪信息：通过 ApplicationReport 的 getTrackingUrl()方法获取应用的跟踪 url，客户端通过该 url 监控进度。

④　应用状态：应用程序的状态可通过 ApplicationReport 的 getYarnApplicationState 方法获取。如果 YarnApplicationState 被设置成了 FINISHED，客户端将会根据 ApplicationReport #getFinalApplicationStatus 来检查应用的状态。在应用失败的情况下，ApplicationReport 的 getDiagnostics 方法是非常有用的。

⑤　如果设置了 AM 支持，客户端可以直接通过 host:rpcport 查询 AM 应用的进度。

2. 编写 AM

应用管理器(AM)是作业的拥有者和管理者。RM 通过客户端得到启动 AM 所有需要的信息和资源，初始化 AM 对象并启动。AM 的启动参数是通过环境提供给它的，当 AM 在容器中被启动的时候，可能与其他的容器共享一个物理主机。AM 启动参数包括：AM 容器的 ContainerId、应用提交时的时间和运行 AM 的 NM 的详细信息。与 RM 的内部交互需要 ApplicationAttemptId，该值可以从 AM 的容器中通过 ApplicationAttemptId 方法获得，并且有方便易用的 API，将从环境获得的值配置到对象中。

```
Map<String, String> envs = System.getenv();
String containerIdString =
```

```
    envs.get(ApplicationConstants.AM_CONTAINER_ID_ENV);
if (containerIdString == null) {
    //container id should always be set in the env by the framework
    throw new IllegalArgumentException(
      "ContainerId not set in the environment");
}
ContainerId containerId = ConverterUtils.toContainerId(containerIdString);
ApplicationAttemptId appAttemptID = containerId.getApplicationAttemptId();
```

在 AM 初始化完成后，启动两个客户端：一个与 ResourceManager 通信，一个与 NodeManagers 通信：

```
AMRMClientAsync.CallbackHandler allocListener = new RMCallbackHandler();
amRMClient = AMRMClientAsync.createAMRMClientAsync(1000, allocListener);
amRMClient.init(conf);
amRMClient.start();
containerListener = createNMCallbackHandler();
nmClientAsync = new NMClientAsyncImpl(containerListener);
nmClientAsync.init(conf);
nmClientAsync.start();
```

AM 必须发送心跳给 RM，保证信息的持续性，表示 AM 正在运行中。AM 与 RM 的连接超时是可配置的，该属性为 YarnConfiguration.RM_AM_EXPIRY_INTERVAL_MS，该值的默认值为 YarnConfiguration.DEFAULT_RM_AM_EXPIRY_INTERVAL_MS。AM 在发送心跳前，需要先向 RM 进程注册。

```
//Register self with ResourceManager
//This will start heartbeating to the RM
appMasterHostname = NetUtils.getHostname();
RegisterApplicationMasterResponse response = amRMClient
  .registerApplicationMaster(appMasterHostname, appMasterRpcPort,
  appMasterTrackingUrl);
```

AM 的注册返回中，包括了最大资源量。可以根据这个信息，判断应用的资源的请求是否能被满足：

```
//Dump out information about cluster capability as seen by the
//resource manager
int maxMem = response.getMaximumResourceCapability().getMemory();
LOG.info("Max mem capabililty of resources in this cluster " + maxMem);
int maxVCores = response.getMaximumResourceCapability().getVirtualCores();
LOG.info("Max vcores capabililty of resources in this cluster " + maxVCores);

//A resource ask cannot exceed the max.
if (containerMemory > maxMem) {
    LOG.info("Container memory specified above max threshold of cluster."
    + " Using max value." + ", specified=" + containerMemory + ", max="
    + maxMem);
    containerMemory = maxMem;
}
if (containerVirtualCores > maxVCores) {
    LOG.info(
      "Container virtual cores specified above max threshold of  cluster."
      + " Using max value." + ", specified=" + containerVirtualCores
      + ", max=" + maxVCores);
    containerVirtualCores = maxVCores;
}
List<Container> previousAMRunningContainers =
  response.getContainersFromPreviousAttempts();
```

```
LOG.info("Received " + previousAMRunningContainers.size()
  + " previous AM's running containers on AM registration.");
```

根据应用的配置，AM 可以申请一批容器来运行它的任务。我们可以计算我们需要多少容器，并申请这些容器：

```
List<Container> previousAMRunningContainers =
    response.getContainersFromPreviousAttempts();
List<Container> previousAMRunningContainers =
    response.getContainersFromPreviousAttempts();
LOG.info("Received " + previousAMRunningContainers.size()
    + " previous AM's running containers on AM registration.");
int numTotalContainersToRequest =
    numTotalContainers - previousAMRunningContainers.size();
//Setup ask for containers from RM
//Send request for containers to RM
//Until we get our fully allocated quota, we keep on polling RM for
//containers
//Keep looping until all the containers are launched and shell script
//executed on them ( regardless of success/failure).
for (int i=0; i<numTotalContainersToRequest; ++i) {
    ContainerRequest containerAsk = setupContainerAskForRM();
    amRMClient.addContainerRequest(containerAsk);
}
```

在方法 setupContainerAskForRM()中，下面的两个信息需要给出。

(1) 资源容量：当前，YARN 支持基于内存的资源请求方式，因此需要明确应用需要的内存大小。

(2) 优先级：当申请容器集时，AM 可能为每个容器集定义不同的优先级。例如，在 MapReduce AM 中，map 可能获得更高的权限容器，而 reduce 获得较低的权限容器。

```
private ContainerRequest setupContainerAskForRM() {
    //setup requirements for hosts
    //using * as any host will do for the distributed shell app
    //set the priority for the request
    Priority pri = Priority.newInstance(requestPriority);

    //Set up resource type requirements
    //For now, memory and CPU are supported so we set memory
    //and cpu requirements
    Resource capability = Resource.newInstance(containerMemory,
      containerVirtualCores);

    ContainerRequest request =
      new ContainerRequest(capability, null, null, pri);
    LOG.info("Requested container ask: " + request.toString());
    return request;
}
```

AM 发送容器申请请求后，容器将会通过 AMRMClientAsync 客户端的事件处理以异步模式启动。当容器分配成功后，处理器将启动一个线程来运行容器：

```
@Override
public void onContainersAllocated(List<Container> allocatedContainers) {
    LOG.info("Got response from RM for container ask, allocatedCnt="
      + allocatedContainers.size());
    numAllocatedContainers.addAndGet(allocatedContainers.size());
    for (Container allocatedContainer : allocatedContainers) {
```

```
        LaunchContainerRunnable runnableLaunchContainer =
          new LaunchContainerRunnable(allocatedContainer,
          containerListener);
        Thread launchThread = new Thread(runnableLaunchContainer);

        //launch and start the container on a separate thread to keep
        //the main thread unblocked
        //as all containers may not be allocated at one go.
        launchThreads.add(launchThread);
        launchThread.start();
    }
}
```

通过心跳，AM 向 RM 汇报作业进度：

```
@Override
public float getProgress() {
    //set progress to deliver to RM on next heartbeat
    float progress =
      (float)numCompletedContainers.get() / numTotalContainers;
    return progress;
}
```

在容器被分配给 AM 后，NMs 上的容器启动线程启动容器：

```
//Set the necessary command to execute on the allocated container
Vector<CharSequence> vargs = new Vector<CharSequence>(5);
//Set executable command
vargs.add(shellCommand);
//Set shell script path
if (!scriptPath.isEmpty()) {
    vargs.add(Shell.WINDOWS?
      ExecBatScripStringtPath : ExecShellStringPath);
}

//Set args for the shell command if any
vargs.add(shellArgs);
//Add log redirect params
vargs.add("1>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stdout");
vargs.add("2>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stderr");

//Get final commmand
StringBuilder command = new StringBuilder();
for (CharSequence str : vargs) {
    command.append(str).append(" ");
}

List<String> commands = new ArrayList<String>();
commands.add(command.toString());

//Set up ContainerLaunchContext, setting local resource, environment,
//command and token for constructor.

//Note for tokens: Set up tokens for the container too. Today, for normal
//shell commands, the container in distribute-shell doesn't need any
//tokens. We are populating them mainly for NodeManagers to be able to
//download any files in the distributed file-system. The tokens are
//otherwise also useful in cases, for e.g., when one is running a
//"hadoop dfs" command inside the distributed shell.
ContainerLaunchContext ctx = ContainerLaunchContext.newInstance(
  localResources, shellEnv, commands, null, allTokens.duplicate(), null);
```

```
  containerListener.addContainer(container.getId(), container);
  nmClientAsync.startContainerAsync(container, ctx);
```

NMClientAsync 对象以及它的时间处理器处理容器事件。包括容器的启动、停止、状态更新、错误发生。在 AM 确定作业完成后，AM 需要通过 AM-RM 客户端对象来注销该作业，然后停止该客户端：

```
try {
   amRMClient.unregisterApplicationMaster(appStatus, appMessage, null);
} catch (YarnException ex) {
   LOG.error("Failed to unregister application", ex);
} catch (IOException e) {
   LOG.error("Failed to unregister application", e);
}
amRMClient.stop();
```

4.10　YARN 服务注册

在分布式环境中，发现与定位服务是一个问题，本节介绍如何将服务注册至集群中。

4.10.1　为什么需要服务注册

服务注册表是部署在 Hadoop 集群上的一个服务，它允许应用注册并与应用建立通信。客户端应用程序可以定位服务，并使用绑定信息连接到服务的网络终端。

客户端如何与 YARN 上已部署的服务交互以及组成服务的组件是什么？服务注册表运行 Hadoop 注册核心服务，减少配置参数、允许核心服务可以被更容易地转移。

服务的注册和发现是在分布式计算领域中一个长期存在的问题。这个方案是为了定位 YARN 集群中的分布式应用，确定与应用交互需要的绑定信息。

4.10.2　配置服务注册

YARN 服务注册服务基于 Apache Zookeeper 创建，通过实现 Hadoop 的 Configuration 类来实现。配置参数决定了注册的客户端和它在 YARN RM 中的部署。服务注册表的默认值都配置在 core-default.xml 中。

修改配置值：如果要修改服务注册表的值，需要在文件 core-site.xml 中重新指定。这可以保证客户端和非 YARN 应用都能读到这些值。

RM 管理在 YARN 容器/应用、临时/应用上用户创建目录和清除记录：

```
<property>
   <description>
如果是 true，YARN 的资源管理器打开了注册功能，反之关闭注册功能。打开的情况下，将创建用户和系统的路径，自动清除容器、作业、attempt 完成时的记录数据。在关闭情况下，用户和系统路径通过其他方式创建，也不会自动清除容器、作业、attempt 的数据
   </description>
   <name>hadoop.registry.rm.enabled</name>
   <value>false</value>
</property>
```

如果 hadoop.registry.rm.enabled 在 core-site.xml 或者 yarn-site.xml 中设置为 true，YARN

RM 将要有下面的行为。

(1) 启动时，RM 创建 root 路径/、/services 和/users。在安全集群中，只限制系统账号可以访问。

(2) 当用户提交应用时：在/users 下创建该用户的目录。

(3) 当容器完成时：删除该容器左右的服务记录。

(4) 当应用临时完成时：删除该应用临时的所有服务记录。

(5) 当应用完成时：删除该应用的所有服务记录。

上面的操作都是异步模式，所以 zookeeper 连接问题不会延迟 RM 工作调度。

如果 hadoop.registry.rm.enabled 属性的值设置为 false，那么，RM 不会与注册器存在交互操作，上面列出的操作也不会发生。root 的根目录可能被其他设置而创建，但是，服务记录的清除不会发生。

4.10.3 安全选项

注册安全可以通过 hadoop.registry.secure 来打开。一旦设置，在 zookeeper 创建权限节点，只有指定的用户和从配置集群的超级账号可以写${hadoop.registry.zk.root}/users。只有超级账号可以访问 root 路径，包括${hadoop.registry.zk.root}/services 和${hadoop.registry.zk.root}/users。对注册器的操作必须认证。读操作对未授权账号也要进行权限验证。

打开安全验证：

```
<property>
   <description>
     Key to set if the registry is secure. Turning it on
     changes the permissions policy from "open access"
     to restrictions on kerberos with the option of
     a user adding one or more auth key pairs down their
     own tree.
   </description>
   <name>hadoop.registry.secure</name>
   <value>false</value>
</property>
```

认证客户端 JAAS 上下文。注册客户端必须指定 JAAS 上下文，用户用于注册认证：

```
<property>
   <description>
     Key to define the JAAS context. Used in secure mode
   </description>
   <name>hadoop.registry.jaas.context</name>
   <value>Client</value>
</property>
```

4.11 小　结

本章围绕 YARN 的基本概念、基本原理、调度模型、编程实践、集群部署与管理进行了介绍与实践。

通过本章的学习，读者应能够基本掌握 YARN 的部署与使用，为后续学习打下基础。

第 5 章

MapReduce

学习目标：

本章带读者一起学习 MapReduce，它是在大数据领域中最为流行且经得住考验的分布式计算框架。MapReduce 对负责逻辑进行高度归约，抽象为 Mapper 和 Reducer 类，复杂逻辑通过理解，转化为符合 MapReduce 函数处理的模式。

本章主要以官方提供的资料为基础，结合实践理解进行讲述。

通过本章的学习，读者应能够掌握 MapReduce 的基本知识，能够编写自己的 MapReduce 程序。

本章要点：

- MapReduce 概述
- Key-Value 结构
- MapReduce 的部署
- MapReduce 的编程接口
- MapReduce 的命令行
- WordCount 的实现

5.1 MapReduce 概述

MapReduce 是一种分布式计算框架，可通过普通廉价的服务器搭建海量且具有极高可扩展性的计算平台。

5.1.1 Hadoop MapReduce

Hadoop MapReduce 是一个软件架构，在数以千计的普通硬件构成的集群中，以平行计算的方式处理海量数据，该计算框架具有很高的稳定性和容错能力。MapReduce job 会划分输入数据集为独立的计算块，这些分块被 map 任务以完全并行、独立模式处理。MapReduce 框架对 maps 的输出进行排序，排序后，数据作为 reduce 任务的输入数据。job 的 input 和 output 数据都存储在 HDFS 文件系统中。计算框架管理作业调度、监控作业、重新执行失败任务。在一个集群中，计算节点和存储节点是相同的，也就是说，MapReduce 框架和 Hadoop 分布式文件系统运行在相同的节点上。这种配置会使得 MR 框架在已有数据的节点上实现本地化的调度任务，从而提升计算性能。MR 框架包括一个 RM 作为管理者，集群中，其他节点上都会部署一个 NodeManager，每类应用会启动一个 MRAppMaster。应用需要指定 input 和 output 的路径，实现接口的抽象函数 map 和 reduce。连同其他参数，构成 job 的配置信息。Hadoop job 客户端提交 job 和配置信息到 RM，RM 负责分发执行包/配置信息到工作节点上，调度任务并监控应用，并把应用状态和诊断信息返回到 job 客户端。

虽然 Hadoop 框架是用 Java 语言实现的，但是，MapReduce 引用可以通过下面的方式以多种语言运行 MapReduce 程序。

(1) Hadoop Streaming：是一个应用程序，相当于用户与 MR 接口之间的 Shell，它能运行其他的语言形式，比如 Shell 程序，作为 map 和 reduce 函数。

(2) Hadoop Pipes：是一个兼容 MapReduce 应用的 C++的 API。

5.1.2 MapReduce 的发展史

MapReduce 的设计思想来源于 Google 的 MapReduce 论文，其核心便是计算并行化，对大文件数据进行分而治之。分布式计算是解决海量数据的有效方法。在面临海量数据的存储、计算挑战时，技术领域有两种截然不同的选择。

第一种选择是：继续提升硬件技术，采用更高性能的计算机服务器。这需要在技术、资金方面投入很多的人力、物力。而且硬件技术的发展遵循摩尔定律，很难应对急速增长的海量数据。因这种选择存在不确定性，所以，这也为探索其他解决方案做了铺垫。

第二种选择是：从软件的角度解决大数据计算问题。单机模式已不能应对 TB 级或更高数据量级的运算，必然寻求多机模式，即现在常说的分布式计算。多机模式的发展也比较早，早期的 P2P 技术就是多机模式的一种。多机模式的核心思想是通过多台计算机协调合作，完成一个计算任务。

分布式计算的要素如下。

(1) 硬件资源：普通服务器即可。一般配置为：硬盘大于或等于 1TB，内存大于或等于 32GB，CPU 大于等于 8 核。

(2) 网络资源：分布式计算是通过网络来协调数据与程序的部署的，在计算过程中也会有大量的中间数据需要在不同的节点上进行传输，因此，网络带宽是分布式集群的一个重要指标。MR 分布式集群一般采用千兆网卡(局域网模式)。

(3) 软件工具，本章介绍的分布式软件为 Hadoop 中的计算框架 MapReduce。

5.1.3　MapReduce 的使用场景

MapReduce 采用 key-value 的编程模式，可以将复杂的业务高度抽象成 map 函数、reduce 函数。MapReduce 常用的处理业务有如下几种。

- 日志解析(数据仓库)：将非结构化数据整理为结构化数据，可以对 HDFS 中的结构化数据通过 hive 管理，经过业务的层域划分，沉淀为数据仓库。ETL 工具的核心功能是源文件数据解析，通过 map 函数，将非规则数据解析成规则数据。日志格式一般可为固定分隔符形式；也可为 Json 格式。例如 Nginx、Apache https 日志，可以格式化为分隔符形式，也可以格式化为 Json 形式。日志格式在企业建设大数据平台时非常重要，合理的日志规范能大大降低 ETL 中的大量重复工作。Json 格式的日志是创建自动化 ETL 工具的最好选择。
- 数据统计：对海量数据的应用，最直接的业务需求即为统计方向。根据 ETL 整理的结构化数据，结合运营的关键指标，用 MapReduce 实现统计逻辑。这需要对程序设计有较高的经验积累，对编程语言也有较高的要求，非研发部门的员工使用 MapReduce 实现统计基本是不可能的。幸运的是，随着 hive 的出现，MapReduce 的使用门槛大大降低了。数据使用人员可以通过 SQL 类的 HiveQL 语句来实现复杂逻辑。但 HiveQL 并不能完全替代 MapRuduce，比如当计算维度不同的统计指标时，只能分别写 HiveQL，造成极大的资源浪费。
- 分析挖掘：这是大数据领域比较深入的部分，用 MapReduce 实现分析模型。用户宽表、用户画像都属于分析挖掘的范畴。Mahout 提供了常用的 30 多种挖掘算法。用户可以基于此方便地实现挖掘任务。

5.2　Key-Value 结构的特点

在介绍 MapReduce 计算框架前，先介绍一下 Key-Value 对数据结构。Key-Value 结构是大数据处理领域中很常见的数据结构。key 具有索引性质，可以快速定位数据；value 数据灵活，字段数、长度变化自由，非常适合根据 key 键值进行查询的需求场景，但对于根据 value 值进行查询的需求，效率会很低，或者工具根本不提供技术接口。

5.2.1　key 的设计

1. 保证 key 的唯一性

如果 key 重复，在 KV 技术中会覆盖已有数据，造成数据丢失或者抛出异常，导致系统

不稳定。比较好的方法如 HBase，会对 key 打时间戳，该时间戳有版本控制，超出最大值后将循环覆盖。在 MR 中的 key 完全由用户控制，key 值相同则将 value 归约为 List 列表。

2. 组成 key 的散列字段具有实际意义

时间字段细化到小时即可，更精确的时间字段对于批处理平台没有多大意义。key 末位最好为 id 字段。key 设计的准则为：根据 idc、业务、规则、id 等，可以拼出 key 值，这是设计 key 的核心所在。不能确定 key，就意味着只能进行范围查询，效率必定低很多。

3. Key 的长度短则性能高

在满足 1、2 的前提下，key 越短越好。在大数据存储中，1 个字节会积累成海量数据。

在 Key-Value 的系统中，key 的设计是最关键环节，糟糕的设计将使得系统性能低下，或者很多需求无法实现。

5.2.2 value 的设计

(1) 把需要的数据信息都写在 value 中。

(2) value 的格式可以采用字段形式，类似数据库的 field；也可以采用 Json 格式。取出 value 后进行格式转换，从而能方便地取到各个字段。

5.3 MapReduce 的部署

我们以 Hadoop 2.7.0 版本为实践版本，读者在使用其他版本时，同样以官方说明为准。单节点部署 Hadoop(Linux，CentOS 6.5)，以伪分布式部署为例，我们将与读者一起，实践 MapReduce 的计算框架。

5.3.1 软件准备

使用 Hadoop 2.7.0，Java 1.7。

不同的 Hadoop 版本需要的 Java 有所不同，需要根据官方文档确定。如最新 2.7 版本的 Hadoop，可用 Java 版本如表 5-1 所示。

表 5-1 可用的 Java 版本

Version	Status	Reported By
oracle 1.7.0_15	Good	Cloudera
oracle 1.7.0_21	Good (4)	Hortonworks
oracle 1.7.0_45	Good	Pivotal
openjdk 1.7.0_09-icedtea	Good (5)	Hortonworks
oracle 1.6.0_16	Avoid (1)	Cloudera
oracle 1.6.0_18	Avoid	Many
oracle 1.6.0_19	Avoid	Many

续表

Version	Status	Reported By
oracle 1.6.0_20	Good (2)	LinkedIn, Cloudera
oracle 1.6.0_21	Good (2)	Yahoo!, Cloudera
oracle 1.6.0_24	Good	Cloudera
oracle 1.6.0_26	Good (2)	Hortonworks, Cloudera
oracle 1.6.0_28	Good	LinkedIn
oracle 1.6.0_31	Good (3, 4)	Cloudera, Hortonworks

5.3.2　配置文件

SSH 及免密登录，配置如下所示：

```
$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
$ chmod 0700 ~/.ssh
```

配置、启动 YARN，配置信息如下所示。

etc/hadoop/mapred-site.xml：

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

etc/hadoop/yarn-site.xml：

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

5.3.3　启动 YARN 守护进程

启动 YARN 守护进程：

```
$ sbin/start-dfs.sh
```

通过 jps 查看是否有 ResourceManager 进程存在。

5.4　MapReduce 的程序结构

MapReduce 程序框架的输入、输出为键值对的形式，需要开发与之对应的 map 与 reduce 方法。

5.4.1 MR 框架的输入和输出

MR 框架仅操作<key, value>对。MR 把 job 的输入当作<key, value>对并产生<key, value>对作为输出，输入和输出的数据类型没有必然关系。key 和 value 类必须被框架进行序列化，有些还需要实现其 Writable 接口。此外，key 类必须实现 WritableComparable 接口，以便容易实现框架排序。MapReduce job 的输入和输出类型如下：

```
(input) <k1, v1> -> map -> <k2, v2> -> combine -> <k2, v2> -> reduce ->
 <k3, v3> (output)
```

5.4.2 WordCount

在详细介绍 MapReduce 之前，让我们先看一个 MapReduce 应用的例子，通过例子理解一下它的工作过程。

WordCount 是一个简单的应用，统计出文件中每个单词的出现次数：

```
public class WordCount {

    public static class TokenizerMapper
      extends Mapper<Object, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context)
          throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class IntSumReducer
      extends Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
          Context context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
```

```
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true)? 0 : 1);
    }
}
```

假设环境变量如下：

```
export JAVA_HOME=/usr/java/default
export PATH=${JAVA_HOME}/bin:${PATH}
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

编译 WordCount.java 并打 JAR 包：

```
$ bin/hadoop com.sun.tools.javac.Main WordCount.java
$ jar cf wc.jar WordCount*.class
```

假设：

/user/joe/wordcount/input 为 HDFS 中的目录。

/user/joe/wordcount/output 为 HDFS 中的目录。

则：

```
$ bin/hadoop fs -ls /user/joe/wordcount/input/
/user/joe/wordcount/input/file01 /user/joe/wordcount/input/file02
$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World Bye World
$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

运行应用：

```
$ bin/hadoop jar wc.jar WordCount /user/joe/wordcount/input
/user/joe/wordcount/output
```

输出为：

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000`
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2`
```

应用可以用-file 选项，后面跟上逗号分隔的路径列表，作为当前任务的工作目录。

以-libjars 选项运行应用，增加 JAR 包到 maps 和 reduces 的 classpath 中。以-archives 选项运行 WordCount 例子，带着-libjars、-files 和-archives 三个选项：

```
bin/hadoop jar hadoop-mapreduce-examples-<ver>.jar wordcount -files
cachefile.txt -libjars mylib.jar -archives myarchive.zip input output
```

其中：

myarchive.zip 表示负载指定目录下的文件。

dir1/dict.txt 和 dir2/dict.txt 两个文件，分别通过符号 dict1 和 dict2 访问。mytar.tgz 放在 tgzdir 的目录中。

下面对实践代码进行重点分析。分别介绍该任务的两个最重要的实现方法。

(1) map 方法：

```
public void map(Object key, Text value, Context context)
  throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
    }
}
```

通过 map 方法实现 Mapper 接口，一次处理一行数据。

比如，可以指定输入为 TextInputFormat 格式。然后可以通过 StringTokenizer 根据空格把一行数据分隔成多个单元，并发送一个 Key-Value 对<word, 1>作为 map 的输出：

```
<Hello, 1>
<World, 1>
<Bye, 1>
<World, 1>
```

(2) reduce 方法：

```
public void reduce(Text key, Iterable<IntWritable> values,
  Context context) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
```

Reducer 的实现中，通过 reduce 方法实现对值的累加，这实现了统计各单词出现的次数。输出形式为：

```
<Bye, 1>
<Goodbye, 1>
<Hadoop, 2>
<Hello, 2>
<World, 2>
```

job 的主方法需要指定各个属性，例如 input/output 的路径，key/value 的类型，input/output 的格式等。之后调用 job.waitForCompletion 来提交 job 并监控它的进展。

5.5 MapReduce 的编程接口

这里将介绍 Reduce 用户接口的细节，帮助用户以合理、高效的方式配置和实现作业 job。Hadoop 官方文档中有对于类接口的详细说明。读者可以自行下载了解并查阅最新版本的变化。MR 中，最重要的两个接口是 Mapper 和 Reducer，可以说，Hadoop 工程师大部分时间都是在写 Mapper 和 Reducer 的实现类。其他重要接口包括 Job、Partitioner、InputFormat、OutputFormat。最后讨论两个框架中有用的属性，例如 DistributedCache、IsolationRunner。

5.5.1 Mapper 接口

Mapper 以 key/value 对作为输入 input，map 方法的输入接口将 input 文件的输入记录转化为内部记录。转换后的中间记录不需要与 input 记录是相同的数据类型。一个 input 数据对可以映射为 0 或多个输出对。Hadoop MR 框架为每个 InputSplit 创建 map 任务。

Mapper 的实现是通过指定 Job.setMapperClass(Class)方法传递给 Job 的。MR 框架据此实例化 Mapper 对象并调用 map 函数处理 InputSplit 中的每个 key/value 对。在作业处理完毕后，用户可以调用 cleanup 函数来处理需要的清理操作，如释放数据库长连接等。map 函数的输出键值对不需要与输入键值对具有相同的数据类型。input 键值对可以映射为空或者其他多种类型的键值对。output 键值对的收集是通过调用 context.write 方法实现的。在 map 之后 reduce 之前的所有的中间数据根据输出键值对的健进行分组聚集，并传递到 reduce 进行处理，最终处理结果汇总到 output 路径中，如果设置 reduce 的输出数为 1，则输出文件有 1 个，达到数据合并的目的。用户可以通过指定 Job.setGroupingComparatorClass(Class)设置 Comparator 来实现对 group 的控制。

map 的输出要经过排序和分区操作后，进入 reduce 处理阶段。对于一个 job 来说，全部的分区数等于全部的 reduce 数。用户可以通过设置用户分区数来控制键值。中间过程的数据结构是以一种简单的(key-len, key, value-len, value)格式存储的，应用可以通过配置控制整个过程。

maps 的数据通常是根据 input 文件的大小来确定的，它是输入文件块数的总和。maps 并行化的适当水平是每个节点 10~100 个 maps。如果我们需要处理 100TB 的文件，并且 HDFS 的 blocksize 为 128MB，那么，我们将要使用 82000 个 map 任务。当然，可以通过设置属性 Configuration.set(MRJobConfig.NUM_MAPS, int)指定 map 数。

5.5.2 Reducer 接口

Reducer 归约中间数据，处理单元是基于 key 的数据集。job 的 Reduces 数量由用户设定的 Job.setNumReduceTasks(int)属性值来确定。Reducer 的实例化对象类通过设置 job 的 Job.setReducerClass(Class)来实现。逻辑上，MR 框架调用 reduce 方法处理 map 的中间输出键值对。中间处理过程包括三个基本的处理阶段：洗牌、排序和归约。

(1) 洗牌。

对多个 map 任务的输出按照不同的分区(Partition)通过网络(HTTP 协议)复制到不同的 reduce 任务节点上，这个过程称为 Shuffle。

(2) 排序。

在这个处理阶段，MR 框架对 Reducer 的数据根据 keys 进行排序。洗牌和排序是同时进行的。

(3) 二次排序。

在 key 排序的基础上，对 value 也进行排序。这种需求就是二次排序。用户可以通过 Job.setSortComparatorClass(Class)指定一个 Comparator。

(4) 归约。

reduce 方法对键值对<key, (list of values)>进行处理。reduce 的输出结果一般都会通过 Context.write(WritableComparable, Writable)写到文件系统中。应用可以使用计数器报告它的统计信息。读者注意：Reducer 的输出是不被排序的。

(5) reduces 数。

reduce 最合适的数字是 0.95 或 1.75×节点数×每个节点上的容器数。

用 0.95 的时候，在 maps 完成时，所有 reduce 可以立即启动并开始传输 map 的输出数据。用 1.75 的时候，快的节点将在完成它们第一轮的 reduces 后启动第二波的 reduce。

(6) Reducer NONE。

没有 Reducer 的 job。在 MapReduce 计算框架中设置 reduce 的个数为 0 是合法的。在这种情况下，map 任务的输出直接写入到文件系统中，该路径是由 FileOutputFormat.setOutputPath(Job, Path)设置的。MR 框架在将数据写入到文件系统之前，不会对 map 的输出进行排序。

5.5.3 Partitioner(分区)

分区是划分 key 空间。Partitioner 控制 map 输出 key 的分区。key 通过哈希函数划分分区，分区的总数等于 job 的 reduce 的任务数。默认的分区器为 HashPartitioner。

5.5.4 Counter(计数器)

计数器是 MR 中被普遍使用的一种报告统计信息的工具。Mapper 和 Reducer 使用计数器来报告应用的统计信息。Hadoop 作业维护若干内置计数器，以描述该作业的各项指标。例如，某些计数器记录已处理的字节数和记录数，使用户可监控各个处理阶段已处理的输入数据量和已产生的输出数据量，例如：

```
FILE: Number of bytes read=76956
FILE: Number of bytes written=3799310
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=83914
HDFS: Number of bytes written=37652
HDFS: Number of read operations=93
HDFS: Number of large read operations=0
HDFS: Number of write operations=2
```

5.5.5 job 工作机理

(1) job 配置信息。

job 提供 MR 的配置对象，job 是用户描述 MR job 到 Hadoop 框架执行的第一接口。它在实现层面上的意义是 MapReduce 类作业在 YARN 框架上运行的封装对象，MR 框架高容错地执行 job。下面的两点需要注意：

① 许多配置文件参数可能标记为 final，不能被更改。

② 大部分 job 参数可以被简单地设置，但还有一些参数是通过复杂的关系继承获得的，用户不能直接找到配置项，需要结合实现，对配置进行逐层分析。

(2)　任务执行和环境变量。

MRAppMaster 在 JVM 上创建 child 进程，独立执行 Mapper/Reducer 任务。子任务继承 MRAppMaster 的环境变量。用户可以通过 mapreduce.{map|reduce}.java.opts 指定其他的子任务的选项，并配置参数，如配置共享库-Djava.library.path=<>等。mapreduce.{map|reduce}.java.opts 参数包含的标签@taskid@代表的值是 MapReduce 任务插入的 taskid。

下面是一个多参数、多子状态的例子，包括设置 JVM GC 日志、免密 JVM Agent，通过 jconsole 连接，可以查看子任务内存、线程和获得线程栈信息。也可是设定 map 和 reduce 任务的最大堆大小，如 512MB 或者 1024MB。

```
<property>
   <name>mapreduce.map.java.opts</name>
   <value>
      -Xmx512M -Djava.library.path=/home/mycompany/lib
        -verbose:gc -Xloggc:/tmp/@taskid@.gc
      -Dcom.sun.management.jmxremote.authenticate=false
        -Dcom.sun.management.jmxremote.ssl=false
   </value>
</property>
<property>
   <name>mapreduce.reduce.java.opts</name>
   <value>
      -Xmx1024M -Djava.library.path=/home/mycompany/lib
        -verbose:gc -Xloggc:/tmp/@taskid@.gc
      -Dcom.sun.management.jmxremote.authenticate=false
        -Dcom.sun.management.jmxremote.ssl=false
   </value>
</property>
```

①　内存管理。

用户/管理员可以为进程、任务指定最大虚拟内存，配置参数为 mapreduce.{map|reduce}.memory.mb。注意，该设置对所有提交的 Job 都有效。mapreduce.{map|reduce}.memory.mb 的值是以为 MB 为单位的，它的值必须大于等于通过命令传给 JVM 的-Xmx 的值。

注意：mapreduce.{map|reduce}.java.opts 只用于配置 MRAppMaster 的子任务。在 map 和 reduce 任务中，通过调整参数提升系统性能，例如调整操作的并发性、访问磁盘的频率；通过文件计数器监控 job 中相关的 map 任务和 reduce 任务的字节数。

②　map 参数。

map 中发射出的数据都被序列化为 buffer，元数据也存储在 buffer 中。序列化缓冲区或者元数据超出了阈值时，缓冲的内容将被存储到硬盘中，而 map 继续往缓冲区输出数据。如果缓冲池满，而写磁盘正在进行，则 map 线程被阻塞。map 任务完成后，所有保存的记录被写进磁盘，且所有在磁盘上的片段合并成一个单一的文件。最小化写磁盘次数可以降低 map 任务的耗时，同时，大的缓冲区也会提供 map 的性能。参数设置方法如表 5-2 所示。

表 5-2　map 参数的设置方法

Name	Type	Description
mapreduce.task.io.sort.mb	int	从 map 中发射数据的序列化缓冲区的大小
mapreduce.map.sort.spill.percent	float	对序列化缓冲区的限制，一旦达到该值，线程将开始溢写操作，即缓冲区的数据写入到硬盘中

注意点：当一个记录比序列化缓冲区大时，MR 框架会触发溢写盘操作，写成一个独立的文件。

③ 洗牌/归约的参数。

如前所述，每个 reduce 通过 HTTP 获取 maps 的输出数据并导入到内存中，然后再合并到磁盘中。如果 map 的输出启动了压缩，那么，中间数据是压缩的，reduce 在内存中进行解压缩。参数的设置方法如表 5-3 所示。

表 5-3 洗牌/归约参数的设置方法

Name	Type	Description
mapreduce.task.io.soft.factor	int	设置在磁盘上同时合并的数据块的数量。它限制了合并过程中可以打开文件的数据和压缩编码解码器的数目。如果超过了该值，则合并操作分多次处理
mapreduce.reduce.merge.inmem.thresholds	int	设置合并 maps 数。在实践中，该值通常设置得非常高(1000)或者让该属性失效(0)，因为在内存中合并比在硬盘中合并时间成本低。该阈值只影响 shuffle 过程中在内存中的合并频率
mapreduce.reduce.shuffle.merge.percent	float	设置合并的内存使用占比。因为 map 的输出不能都放在内存中，该值设置得高，可以降低获取和合并的并行度。相反地，当该值为 1 时，对 reduce 是高效的，reduce 的输入都可以放在内存中。该参数只影响 shuffle 过程中在内存中合并的频度
mapreduce.reduce.shuffle.input.buffer.percent	float	设置 Mapreduce.reduce.java.opts 所使用内存的百分比。该内容用于在 shuffle 过程中存储 map 的输出。一些内存是为 MR 框架预留的，一般来说，这些内存足以存储 map 的输出
mapreduce.reduce.input.buffer.percent	float	设置保存 reduce 中 map 输出的内存使用占比。reduce 开始后，map 输出将要合并到硬盘。默认情况下，在 reduce 申请到最大资源前，所有 map 的输出都写入磁盘。对于少数内存密集型的 reduce，该值应增加，以避免对磁盘的多次访问

其他注意点：

- 如果一个 map 输出的大小超过了开辟内存的 25%，这些输出数据将跳过存储在内存中的阶段，直接写入到磁盘。
- 当有 combiner 时，合并操作在所有的 map 输出前就开始。当合并阈值和缓冲区可能不足时，combine 则溢出写盘。在很多情况下，分配资源合并 map 的输出比增加缓冲区大小更有效。
- 在内存中合并 map 的输出并写到磁盘后，再开始 reduce 处理。

④　配置参数(Configured Parameters)。

对每一个 job 的配置属性如表 5-4 所示。

表 5-4　对每一个 job 的配置属性

Name	Type	Description
mapreduce.job.id	String	job 的 ID
mapreduce.job.jar	String	JAR 包的目录
mapreduce.job.local.dir	String	job 的共享空间
mapreduce.task.id	String	任务的 id
mapreduce.task.attempt.id	String	任务 attempt 的 id
mapreduce.task.is.map	boolean	是否是一个 map 任务
mapreduce.task.partition	int	job 中任务的 id
mapreduce.map.input.file	String	map 读取的文件名
mapreduce.map.input.start	long	map 输入分片的偏移量
mapreduce.map.input.length	long	map 输出分片中的字节总数
mapreduce.task.output.dir	String	任务的临时输出目录

注意：在 streaming job 执行的过程中，MapReduce 的名称参数是被转换的，规则为“点”变成“下划线”。例如，mapreduce.job.id 变成 mapreduce_job_id，mapreduce.job.jar 变成 mapreduce_job_jar。

⑤　任务日志(Task Logs)。

标准输出和错误输出流及 task 的日志在 NM 节点上产生，目录为${HADOOP_LOG_DIR}/userlogs。

⑥　分发库(Distributing Libraries)。

分布式缓存可以被用于分发 JAR 包和 map/reduce 任务中要依赖的本地库。Java 虚拟机把 java.library.path 和 LD_LIBRARY_PATH 加到它的工作目录中。因此，缓存库可以通过 System.loadLibrary 或 System.load 加载。

5.5.6　任务提交和监控(Job Submission and Monitoring)

job 是用户在 RM 上编写程序的最基础接口。job 提供了多个模块，分别提交应用、跟踪进度、访问 tasks 组件报告和日志，获得 RM 的状态信息等。

job 提交的过程如下。检查 job 的输入和输出路径；计算 job 的输入分片；如果需要的话，计算 DistributedCache 所需要的账号信息；复制 job 的 JAR 和配置信息到 HDFS 文件系统目录中；提交任务到 RM 并监控它的状态。

job 历史文件被写到用户指定的目录中：mapreduce.jobhistory.intermediate-done-dir 和 mapreduce.jobhistory.done-dir，该属性默认为 job 的输出路径。在指定目录执行下面的命令，用户可以查看历史日志的摘要信息：mapred job -history output.jhist。该命令将输出 job 的详

细信息、失败和 kill 提示信息。关于 job 的更多的信息，例如成功任务和任务 attempt，可以通过下面的命令进行查看：$ mapred job -history all output.jhist。

(1) job 控制。

用户可能需要串联多个 MR 任务来合作完成负责的业务逻辑，这些逻辑不能通过一个 MR 实现。这是非常简单的，因为上一个 job 的输出写入分布式文件系统，接着作为下一个 job 的输入数据。当然，这需要确认上一个 job 是否完成(成功/失败)。在这种情况下，各种任务控制选项有：

- Job.submit()：提交 job 到集群，并马上返回。
- Job.waitForCompletion(boolean)：提交 job 到集群，并等待 job 完成。

(2) job 的输入。

InputFormat 描述了 MR job 的输入。

MR 框架关于 InputFormat 的描述是：验证 job 的 input 信息，划分输入数据为多个分片，每个分片都将分配独立执行的 Mapper 任务。提供 RecordReader 的实现，用于收集 Mapper 的 InputSplit 数据。

输入分片(InputSplit)提供每个独立 Mapper 处理的数据。InputSplit 是 input 的一个面向字节的视图，它可以被 RecordReader 对象转化为面向记录的视图。FileSplit 是默认的 InputSplit，Mapreduce.map.input.file 设置输入路径。

RecordReader 从 InputSplit 读取<key, value>数据对。RecordReader 将字节视图转换为记录视图。

(3) job 的输出。

OutputFormat 描述 MR 应用的输出格式。其职责包括：

- 检查 job 的输出；例如检查 output 目录是否已经存在。
- 提供对 job 的输出的 RecordWriter，作为写者。输出文件被存储在文件系统中。

TextOutputFormat 是默认的 OutputFormat。OutputCommitter 是一个 MR job 输出的委托。

MR 框架依靠 OutputCommitter，以实现下列目的。

① 初始化阶段建立 job。例如，在 job 的初始化阶段，创建临时输出目录。任务处于 PREP 阶段时，job 的建立是由一个独立的任务来完成的，一旦这个 setup 任务完成后，job 将被移到 RUNNING 阶段。

② job 完成后的清理工作。例如在 job 完成后，删除链式的输入目录。job 的清理工作是在 job 完成后，由一个独立的任务完成的。在 job 清除工作完成后，job 的状态将变为 SUCCEDED/FAILED/KILLED。

③ 创建任务的临时输入。

④ 检查 job 是否需要提交，避免提交不需要提交的 job。

⑤ 提交任务的输出。一旦任务完成后，如果需要，任务将提交它的输出。如果任务失败或者被 killed，那么，输出将被清理。如果清理不能执行，一个独立的任务将被启动，该任务具有相同的 attempt-id，专门做清理工作。

FileOutputCommitter 是默认的 OutputCommitter。job 在 NM 上创建/清理 map 或 reduce 的容器。job 清理、task 清理任务和 Job 创建任务具有高优先级。

5.5.7　任务的辅助文件(Task Side-Effect Files)

应用中，组件任务需要创建或写数据到辅助文件中，辅助文件是与 job 数据文件不同的文件。在有些情况下，当同时运行两个 Mapper 或者 Reducer 实例时，就可能遇到辅助文件冲突，比如两个实例同时打开并写相同的文件。

application-writer 可以根据作业需要，在${mapreduce.task.output.dir}中创建辅助文件，这个目录可以通过 FileOutputFormat.getWorkOutputPath(Conext)来获取。

注意：在 MR 框架中，task 任务的${mapreduce.task.output.dir}值会变成${mapreduce.output.fileoutputformat.outputdir}/_temporary/_{$taskid}。利用此属性，任何一个任务可以通过 FileOutputFormat.getWorkOutputPath(Conext)来获取辅助文件路径，并创建、写入数据。对于 Reducer 是空的任务，map 的输出直接写到 HDFS 中。

5.5.8　提交作业到队列

用户提交的任务都会分配到队列中，队列作为 job 的收集者，为运行 MR 框架提供特殊的功能。例如，使用 ACLs 来控制用户提交的 job 到哪个队列。Hadoop 运行后会有一个默认的队列，称为 default。队列的名称通过 mapreduce.job.queuename 属性进行设置。应用调度器的容量调度器支持多个队列。

一个 job 需要指定要提交到哪个队列，通过 mapreduce.job.queuename 属性，或者通过 API:onfiguration.set(MRJobConfig.QUEUE_NAME, String)来设置队列名。如果一个 job 提交时没有指定队列，那么，默认提交到 default 队列中。

5.5.9　MR 中的计数器(Counters)

Counters 接口提供了全局计数器，在 MR 框架和应用中被定义使用。应用可以通过 map 和 reduce 方法中的 Counters.incrCounter(Enum, long)或者 Counters.incrCounter(String, String, long)更新计数器的数据。这些计数器被 MR 框架全局统计。

5.5.10　Profiling

程序概要分析是一个获取内置 Java 框架的应用程序。用户可以指定系统是否为 job 中的 task 收集轮廓信息，该功能通过 mapreduce.task.profile 属性设置。该值也可以通过 API 来设置 Configuration.set(MRJobConfig.TASK_PROFILE, boolean)。如果设置为 true，那么任务的轮廓信息可用，反之不可用。job 的轮廓信息存储在用户的日志目录中，默认情况下，job 的程序概要功能是关闭的。

当用户需要进行程序概要分析时，可以通过配置 mapreduce.task.profile.{maps|reduces}属性来设置程序概要分析的作用范围。该值也可以通过 API 设置：Configuration.set(MRJobConfig.NUM_{MAP|REDUCE}_PROFILES, String)。默认情况下，指定的范围是 0~2。用户也可以指定配置文件轮廓参数，设置配置文件的属性 mapreduce.task.profile.params。该值也可以通过 API 来设置：Configuration.set(MRJobConfig.TASK_PROFILE_PARAMS, String)。如果字符串包括%s，它将会被 task 的输出文件名代替，该参数以命令行的模式传

递给子 JVM。Profile 的默认值是-agentlib:hprof=cpu=samples,heap=sites,force=n,thread=y,verbose=n,file=%s。

5.5.11 Debugging

MR 框架提供了脚本调试工具。当一个 MR 任务失败时，用户可以运行 debug 脚本，来处理 task 日志。Script 可以访问 stdout、stderr 输出、系统日志、job 配置。stdout 和 stderr 的输出在控制台，输出即可作为诊断信息，这也是 job UI 的一部分。

下面的部分，我们将介绍如何提交一个 job 的 debug 脚本。脚本文件需要分布式并且提交给 MR 计算框架。

如何提交 debug 脚本？一种比较常用提交 debug 脚本的方式是设置属性 mapreduce.map.debug.script 和 mapreduce.reduce.debug.script，对 map 和 reduce 任务独立调试。该属性也可以通过 API 来设置：Configuration.set(MRJobConfig.MAP_DEBUG_SCRIPT, String) 和 Configuration.set(MRJobConfig.REDUCE_DEBUG_SCRIPT, String)。在流模式中，debug script 可以通过命令行选项-mapdebug 和-reducedebug 来设置，对 map 和 reduce 任务分别进行调试。

debug 脚本的参数是任务的 stdout、stderrr、syslog 和 jobconf。debug 命令是：

```
$script $stdout $stderr $syslog $jobconf
```

5.5.12 job Outputs

作业可以通过 FileOutputFormat.setCompressOutput(Job, boolean)来控制 job 输出的压缩器，通过 FileOutputFormat.setOutputCompressorClass(Job, Class)指定输出的解压器。

如果 job 的输出以 SequenceFileOutputFormat 的形式存储，压缩器选择 SequenceFile.CompressionType(例如 RECORD/BLOCK，默认为 RECORD)。

5.5.13 忽略坏记录(Skipping Bad Records)

Hadoop 提供忽略坏记录、大小写选项功能，可以通过 SkipBadRecords 类对 map 的输入进行坏记录忽略、大小写设置。该属性在 map 宕机时非常有用。往往是因为 map 函数中的程序 bug 导致。对于程序 bug，用户必须修复。但有的情形下，可能需要忽略这些错误的数据。bug 可能是第三方库中的(例如，没有源代码)。在这种情况下，有些任务是永远无法成功完成的，进而导致 job 失败。通过该属性，丢失一小部分坏数据，从业务的角度讲，对于某些应用来说，是可以接受的。默认情况下，该属性是关闭的。可以通过 SkipBadRecords.setMapperMaxSkipRecords(Configuration, long)和 SkipBadRecords.setReducerMaxSkipGroups(Configuration, long)打开。通过该属性，在 map 失败几次后，MR 计算框架进入忽略模式。job 的计数器记录了忽略的坏记录的个数。

增加忽略大小写参数-Dwordcount.case.sensitive：

```
$ bin/hadoop jar wc.jar WordCount2 -Dwordcount.case.sensitive=false
/user/joe/wordcount/input /user/joe/wordcount/output -skip
/user/joe/wordcount/patterns.txt
```

输出结果为：

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000
bye 1
goodbye 1
hadoop 2
hello 2
horld 2
```

5.6　MapReduce 的命令行

MapReduce 可以通过命令行的形式进行任务调度与管理，本节将介绍这些命令的具体使用。

5.6.1　概述

所有的 MapReduce 命令都通过 bin/mapred 脚本调用。运行不带参数的 mapred 脚本可打印所有命令的描述：

```
Usage: mapred [--config confdir] [--loglevel loglevel] COMMAND
      where COMMAND is one of:
  pipes              run a Pipes job
  job                manipulate MapReduce jobs
  queue              get information regarding JobQueues
  classpath          prints the class path needed for running
                    mapreduce subcommands
  historyserver       run job history servers as a standalone daemon
  distcp <srcurl> <desturl> copy file or directories recursively
  archive -archiveName NAME -p<parent path><src>*<dest>
    create a hadoop archive
  hsadmin             job history server admin interface
```

命令选项如表 5-5 所示。

表 5-5　mapred 的命令选项

COMMAND_OPTIONS	Description
SHELL_OPTIONS	Shell 选项通用集合
GENERIC_OPTIONS	支持的命令选项集
COMMAND COMMAND_OPTIONS	带参数的命令

Hadoop 有一套用户命令集，与运行类时的选项相似，下面将进行介绍。

5.6.2　用户命令(User Commands)

(1)　archive：创建 Hadoop 档案。

用法：hadoop archive -archiveName name -p <parent>[-r<replication factor>]<src>*<dest>

(2)　classpath：设置 Hadoop JAR 和需要依赖的其他库。

用法：mapred classpath

(3)　Distcp：递归地复制文件或目录。

用法：hadoop distcp hdfs://nn1:8020/foo/bar \

(4) job：与 MapReduce job 进行交换的命令。

用法：mapred job | [GENERIC_OPTIONS] | [-submit <job-file>] | [-status <job-id>] | [-counter <job-id> <group-name> <counter-name>] | [-kill <job-id>] | [-events <job-id> <from-event-#> <#-of-events>] | [-history [all] <jobOutputDir>] | [-list [all]] | [-kill-task <task-id>] | [-fail-task <task-id>] | [-set-priority <job-id> <priority>]

命令选项如表 5-6 所示。

表 5-6 job 的命令选项

COMMAND_OPTION	Description
-submit job-file	提交 job
-status job-id	打印 map 和 reduce 完成的百分比、所有 job 的计数器
-counter job-id group-name counter-name	打印计数器的值
-kill job-id	kill 指定的 job
-history [all]jobOutputDir	打印 job 的详细信息，失败、被 kill 的详细信息
-list [all]	显示尚未完成的 jobs，-list all 显示所有的 jobs
-kill-task task-id	kill 一个 task 任务
-fail-task task-id	使指定的 task 任务失败，失败的任务计入失败尝试测试
-set-priority job-id priority	修改 job 的优先级，可选择的优先级有：VERY_HIGH、HIGH、NORMAL、LOW、VERY_LOW

(5) pipes：管道。

用法：mapred pipes [-conf <path>] [-jobconf <key=value>, <key=value>, ...] [-input <path>] [-output <path>] [-jar <jar file>] [-inputformat <class>] [-map <class>] [-partitioner <class>] [-reduce <class>] [-writer <class>] [-program <executable>] [-reduces <num>]

命令选项如表 5-7 所示。

表 5-7 pipes 的命令选项

COMMAND_OPTION	Description
-conf path	指定 job 的配置文件
-jobconf key=value, key=value, ...	增加或者覆盖 job 的配置文件
-input path	输入目录
-output path	输出目录
-jar jar file	JAR 包名
-inputformat class	InputFormat 类

续表

COMMAND_OPTION	Description
-map class	Java map 类
-partitioner class	Java Partitioner 类
-reduce class	Java Reduce 类
-writer class	Java RecordWriter 类
-program executable	执行 URI
-reduces num	Reducer 并行数

(6)　version：打印 MapReduce 的版本。

用法：mapred version。

5.6.3　管理员命令(Administration Commands)

(1)　historyserver：启动 JobHistoryServer。

用法：mapred historyserver

(2)　hsadmin：运行 MapReduce hsadmin 客户端，来执行 JobHistoryServer 的管理命令。

用法：mapred hsadmin [-refreshUserToGroupsMappings] | [-refreshSuperUserGroups-Configuration] | [-refreshAdminAcls] | [-refreshLoadedJobCache] | [-refreshLogRetentionSettings] | [-refreshJobRetentionSettings] | [-getGroups [username]] | [-help [cmd]]

命令选项如表 5-8 所示。

表 5-8　hsadmin 的命令选项

COMMAND_OPTION	Description
-refreshUserToGroupsMappings	更新 user-to-groups 的映射
-refreshSuperUserGroupsConfiguration	更新超级代理组的映射
-refreshAdminAcls	更新对历史 job 服务的管理员的访问控制列表
-refreshLoadedJobCache	更新 job 历史服务的缓冲区
-refreshJobRetentionSettings	更新 job 历史数据的过期时间、清理周期
-refreshLogRetentionSettings	更新日志保留时长和日志保留检查周期
-getGroups [username]	获取用户所属的群组
-help [cmd]	获取指定命令的帮助说明

执行命令 sbin/mr-jobhistory-daemon.sh start historyserver：

```
[hadoop@hadoop-nn hadoop]$ sbin/mr-jobhistory-daemon.sh start historyserver
starting historyserver, logging to
/usr/local/hadoop/hadoop-2.7.2/logs/mapred-hadoop-historyserver-hadoop
-nn.out
```

5.6.4 YARN-MapReduce 的部署

YARN 配置文件修改：vim yarn-site.xml。

```
<configuration>
    <!-- Site specific YARN configuration properties -->
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.auxservices.mapreduce.shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>
    <property>
        <name>yarn.resourcemanager.address</name>
        <value>hadoop-rm:8032</value>
    </property>
    <property>
        <name>yarn.resourcemanager.scheduler.address</name>
        <value>hadoop-rm:8030</value>
    </property>
    <property>
        <name>yarn.resourcemanager.resource-tracker.address</name>
        <value>hadoop-rm:8031</value>
    </property>
    <property>
        <name>yarn.resourcemanager.admin.address</name>
        <value>hadoop-rm:8033</value>
    </property>
    <property>
        <name>yarn.resourcemanager.webapp.address</name>
        <value>hadoop-rm:8088</value>
    </property>
</configuration>
```

修改 yarn-env.sh 中的 JAVA_HOME，vim hadoop-env.sh。

```
# The java implementation to use.
#export JAVA_HOME=${JAVA_HOME}
export
JAVA_HOME="/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.71-2.b15.el7_2.x86_64"
```

MapReduce 配置文件修改：vim mapred-site.xml。

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>mapreduce.jobhistory.address</name>
        <value>hadoop-nn:10020</value>
    </property>
    <property>
        <name>mapreduce.jobhistory.webapp.address</name>
        <value>hadoop-nn:19888</value>
    </property>
</configuration>
```

启动分布式系统(YARN)。

启动 YARN，在/home/hadoop/hadoop 下执行：

```
[hadoop@hadoop-nn hadoop]$ sbin/start-yarn.sh
starting yarn daemons
starting resourcemanager, logging to
/usr/local/hadoop/hadoop-2.7.2/logs/yarn-hadoop-resourcemanager-hadoop
-nn.out
hadoop-nn: starting nodemanager, logging to
/usr/local/hadoop/hadoop-2.7.2/logs/yarn-hadoop-nodemanager-hadoop
-nn.out
[hadoop@hadoop-nn hadoop]$ jps
1568 SecondaryNameNode
1360 DataNode
3504 ResourceManager
3621 NodeManager
3932 Jps
1246 NameNode
2495 JobHistoryServer
```

5.7　WordCount 的实现

WordCount 是 Hadoop 的例子中的代码，比较简单，是学习 MapReduce 编程的经典案例，主要分为 job 设置、Map 过程、Combine 过程、Reduce 过程，并且 Combine 与 Reduce 实例化相同的 Reducer 对象。

下面先看下 WordCount 类：

```
public class WordCount {

    public static class TokenizerMapper
      extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context)
          throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }
    public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
          Context context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
```

```
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs =
          new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: wordcount <in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true)? 0 : 1);
    }
}
```

主要实现 Mapper、Reducer 类，它们的继承关系为：

```
public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{}
public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {}
```

其中斜体的数据类型必须严格一致，否则报错。

(1) 创建 job：

```
Job job = new Job(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
```

其中的 job 为任务实例，是 Mapreduce 任务的具体对象。

setMapperClass：设置 Mapper 类名。

setCombinerClass：设置 Cobiner 类名，可以与 Reducer 类名相同。

setReducerClass：设置 Reducer 类名。

setOutputKeyClass：设置输出 key 的数据类型。

setOutputValueClass：设置输出 value 的数据类型。

FileInputFormat.addInputPath：设置 MapReduce 任务的输入路径。

FileOutputFormat.setOutputPath：设置 MapReduce 任务的输出路径。

(2) map 函数：

```
public void map(Object key, Text value, Context context)
  throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
```

```
            word.set(itr.nextToken());
            context.write(word, one);
        }
```

参数说明：Object key 是输入的 key，一般由 Mapper 自动生成；Text value 是输入的 value，通常指具体的行内容。Context context 是上下文变量，传递从 map 到 reduce 的 key、value。

WoudCount 的 map 函数逻辑：首先对行数据进行分词，以空格作为分词标签，将每个词构建成最简单的键值对(word，1)，并将该值传递给 reduce 函数。

(3)　reduce 函数：

```
public void reduce(Text key, Iterable<IntWritable> values,
  Context context) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
```

参数说明：Text key 是 reduce 的 key，从 combine 传递过来，即各个 word；Iterable <IntWritable> values 是 value 列表，表示从各个 combine 归约来的对应该 key 的 value 值集合，其形式为(word，n)其中 n≥1。Context context 表示上下文变量，指定输出到输出目录的 key-value 值。

WoudCount 的 reduce 函数逻辑：根据 word 进行 value 的累加，然后将 word 和计算后的 sum 作为输出。

上传要处理的文件：将 Hadoop 所有的配置文件放到/home/hadoop/wordcount/input 中。

在 HDFS 中创建目录：

```
hdfs dfs -mkdir /home
hdfs dfs -mkdir /home/hadoop
hdfs dfs -mkdir /home/hadoop/wordcount
hdfs dfs -mkdir /home/hadoope/wordcount/input
hdfs dfs -put /usr/local/hadoop/hadoop/etc/hadoop/*
/user/hadoop/wordcount/input/
```

查看是否上传成功：

```
hdfs dfs -ls /home/yourname/wordcount/input
```

执行方法及输出信息：

```
[hadoop@hadoop-nn hadoop]$ yarn jar
share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar  wordcount
/user/hadoop/wordcount/input/*  /user/hadoop/wordcount/output01
16/04/09 17:14:53 INFO client.RMProxy: Connecting to ResourceManager at
hadoop-rm/10.172.89.217:8032
16/04/09 17:14:54 INFO input.FileInputFormat:
Total input paths to process:30
16/04/09 17:14:54 INFO mapreduce.JobSubmitter: number of splits:30
16/04/09 17:14:55 INFO mapreduce.JobSubmitter: Submitting tokens for job:
job_1460192892083_0002
16/04/09 17:14:55 INFO impl.YarnClientImpl: Submitted application
application_1460192892083_0002
16/04/09 17:14:55 INFO mapreduce.Job: The url to track the job:
http://hadoop-rm:8088/proxy/application_1460192892083_0002/
```

```
16/04/09 17:14:55 INFO mapreduce.Job: Running job: job_1460192892083_0002
16/04/09 17:15:10 INFO mapreduce.Job: Job job_1460192892083_0002 running in
uber mode : false
16/04/09 17:15:10 INFO mapreduce.Job:  map 0% reduce 0%
16/04/09 17:15:43 INFO mapreduce.Job:  map 20% reduce 0%
16/04/09 17:16:16 INFO mapreduce.Job:  map 40% reduce 0%
16/04/09 17:16:44 INFO mapreduce.Job:  map 40% reduce 13%
16/04/09 17:16:48 INFO mapreduce.Job:  map 57% reduce 13%
16/04/09 17:16:52 INFO mapreduce.Job:  map 57% reduce 19%
16/04/09 17:17:16 INFO mapreduce.Job:  map 73% reduce 23%
16/04/09 17:17:20 INFO mapreduce.Job:  map 73% reduce 24%
16/04/09 17:17:42 INFO mapreduce.Job:  map 90% reduce 24%
16/04/09 17:17:44 INFO mapreduce.Job:  map 90% reduce 30%
16/04/09 17:17:58 INFO mapreduce.Job:  map 100% reduce 30%
16/04/09 17:18:00 INFO mapreduce.Job:  map 100% reduce 100%
16/04/09 17:18:01 INFO mapreduce.Job: Job job_1460192892083_0002 completed
successfully
16/04/09 17:18:01 INFO mapreduce.Job: Counters: 50
File System Counters
FILE: Number of bytes read=76956
FILE: Number of bytes written=3799310
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=83914
HDFS: Number of bytes written=37652
HDFS: Number of read operations=93
HDFS: Number of large read operations=0
HDFS: Number of write operations=2
Job Counters
Killed map tasks=1
Launched map tasks=31
Launched reduce tasks=1
Data-local map tasks=31
Total time spent by all maps in occupied slots (ms)=828078
Total time spent by all reduces in occupied slots (ms)=102418
Total time spent by all map tasks (ms)=828078
Total time spent by all reduce tasks (ms)=102418
Total vcore-milliseconds taken by all map tasks=828078
Total vcore-milliseconds taken by all reduce tasks=102418
Total megabyte-milliseconds taken by all map tasks=847951872
Total megabyte-milliseconds taken by all reduce tasks=104876032
Map-Reduce Framework
Map input records=2168
Map output records=8074
Map output bytes=108479
Map output materialized bytes=77130
Input split bytes=3954
Combine input records=8074
Combine output records=4064
Reduce input groups=1600
Reduce shuffle bytes=77130
Reduce input records=4064
Reduce output records=1600
Spilled Records=8128
Shuffled Maps =30
Failed Shuffles=0
Merged Map outputs=30
GC time elapsed (ms)=18827
CPU time spent (ms)=18640
Physical memory (bytes) snapshot=6996582400
```

```
Virtual memory (bytes) snapshot=64411631616
Total committed heap usage (bytes)=5451743232
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
Bytes Read=79960
File Output Format Counters
Bytes Written=37652
```

其中：

Map input records=2168 表示输入路径中的文件的行数，检查方法：

```
[hadoop@hadoop-nn hadoop]$ hdfs dfs -cat /user/hadoop/wordcount/input/*
|wc -l
2168
```

Map output records=8074 表示输出的文件中的函数，此为中间值。该值与 combine 的 input 值理论上相等。

Combine input records=8074 表示 map 后的行数，此为中间值。

Combine output records=4064 表示 combine 输出的行数，此为中间值，输出到 reduce 阶段，该值理论上与 reduce 的 input 数值相等。

Reduce input records=4064 表示 reduce 的输入行数。

Reduce output records=1600 表示归约后的行数，也是写到输出文件的行数，检查方法：

```
[hadoop@hadoop-nn hadoop]$ hdfs dfs -cat /user/hadoop/wordcount/output01/*
|wc -l
1600
```

输出结果片段：

```
[hadoop@hadoop-nn hadoop]$ hdfs dfs -cat
/user/hadoop/wordcount/output01/part-r-00000 | more
!=      3
""      6
"".     4
"$HADOOP_CLASSPATH"     1
"$JAVA_HOME"    2
"$YARN_HEAPSIZE"        1
"$YARN_LOGFILE" 1
"$YARN_LOG_DIR" 1
"$YARN_POLICYFILE"      1
"*"     18
"AS     25
"Error: 1
"License");     25
"alice,bob      18
"console"       1
"dfs"   3
"hadoop.root.logger".  1
"jks". 4
"jvm"   3
"mapred"        3
```

增加忽略大小写参数-Dwordcount.case.sensitive：

```
[hadoop@hadoop-nn hadoop]$ yarn jar
share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar  wordcount
-Dwordcount.case.sensitive=false /user/hadoop/wordcount/input/*
/user/hadoop/wordcount/output02
```

该任务可以通过 WebUI 来查看，默认网址为：

```
http://{yarn-host}:8088
```

页面截图如图 5-1 所示。

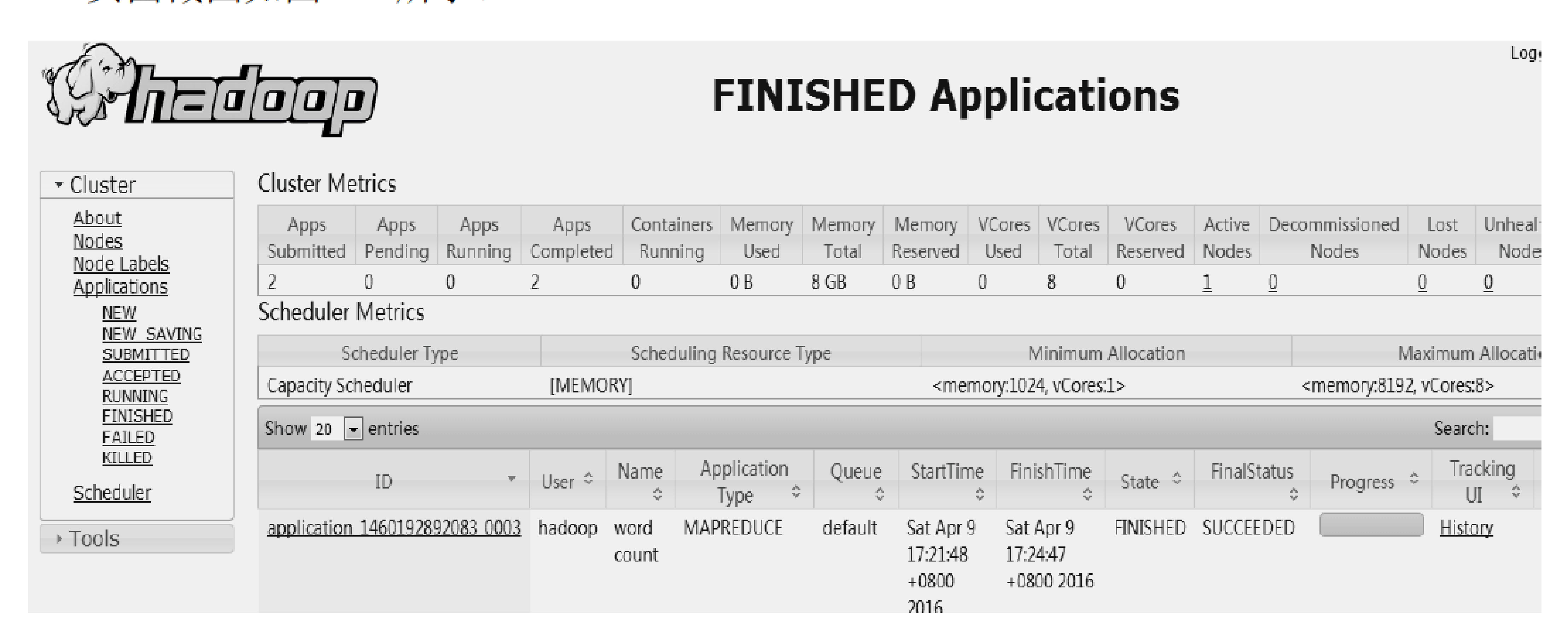


图 5-1 MapReduce 任务的效果

调度器截图如图 5-2 所示。

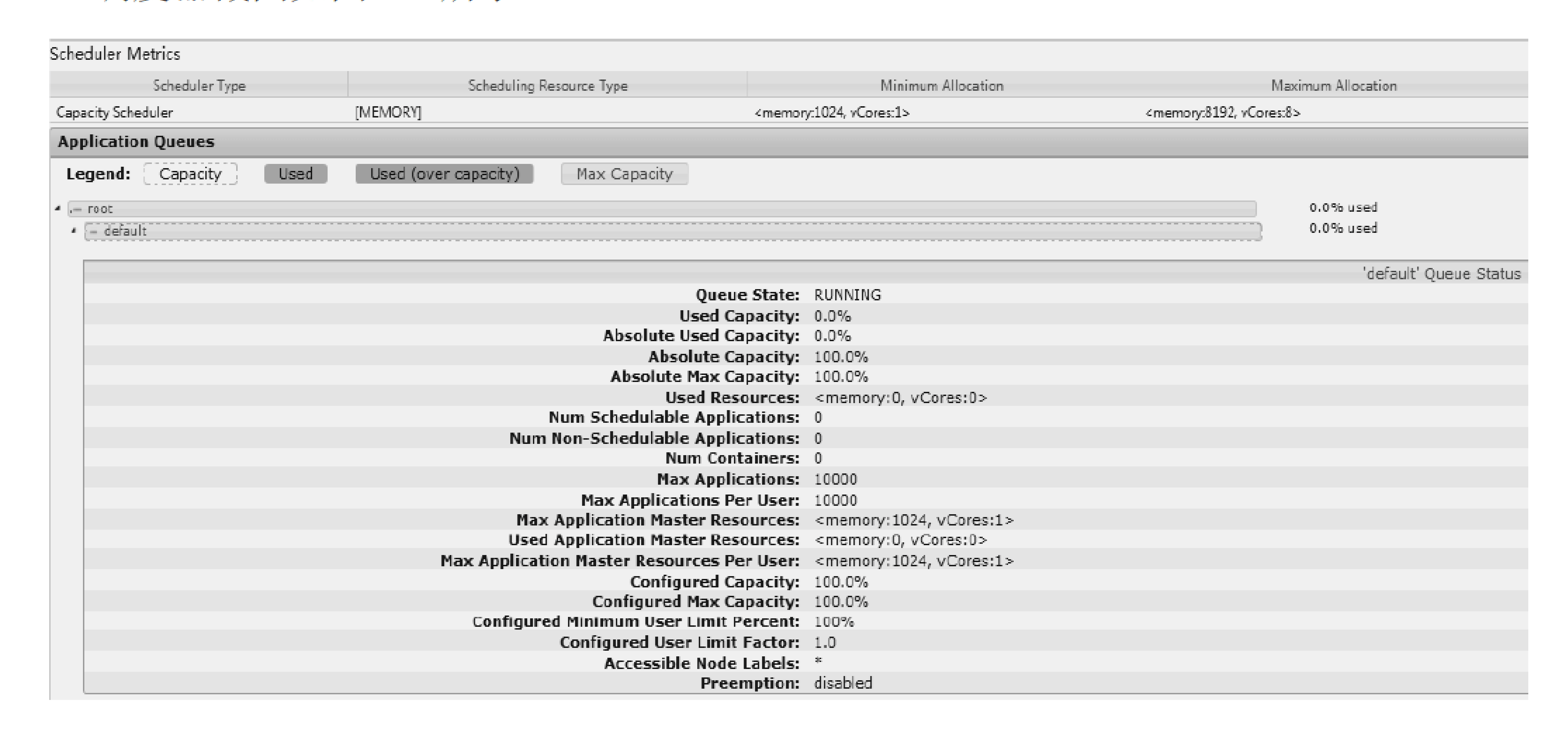


图 5-2 调度器的效果

查看该任务的日志，则需要开启 jobhistory 服务，做内外网 IP 端口映射，并且在本地配置 hosts。

启动服务：

```
[hadoop@hadoop-nn hadoop]$ sbin/mr-jobhistory-daemon.sh start historyserver
```

通常集群搭建设置的是内容 IP，如果要外网访问 Web 服务，就需要进行内网、外网的防火墙映射：

```
iptables -A PREROUTING -t nat -p tcp --dport 19888 -j DNAT --to-destination
10.172.89.xxx:19888
iptables -A PREROUTING -t nat -p tcp --dport 8088 -j DNAT --to-destination
10.172.89.xxx:8088
```

Hosts 配置：

```
123.56.76.xxx hadoop-rm hadoop-nn （两者为配置文件中的配置信息）
```

日志查看界面如图 5-3 所示。

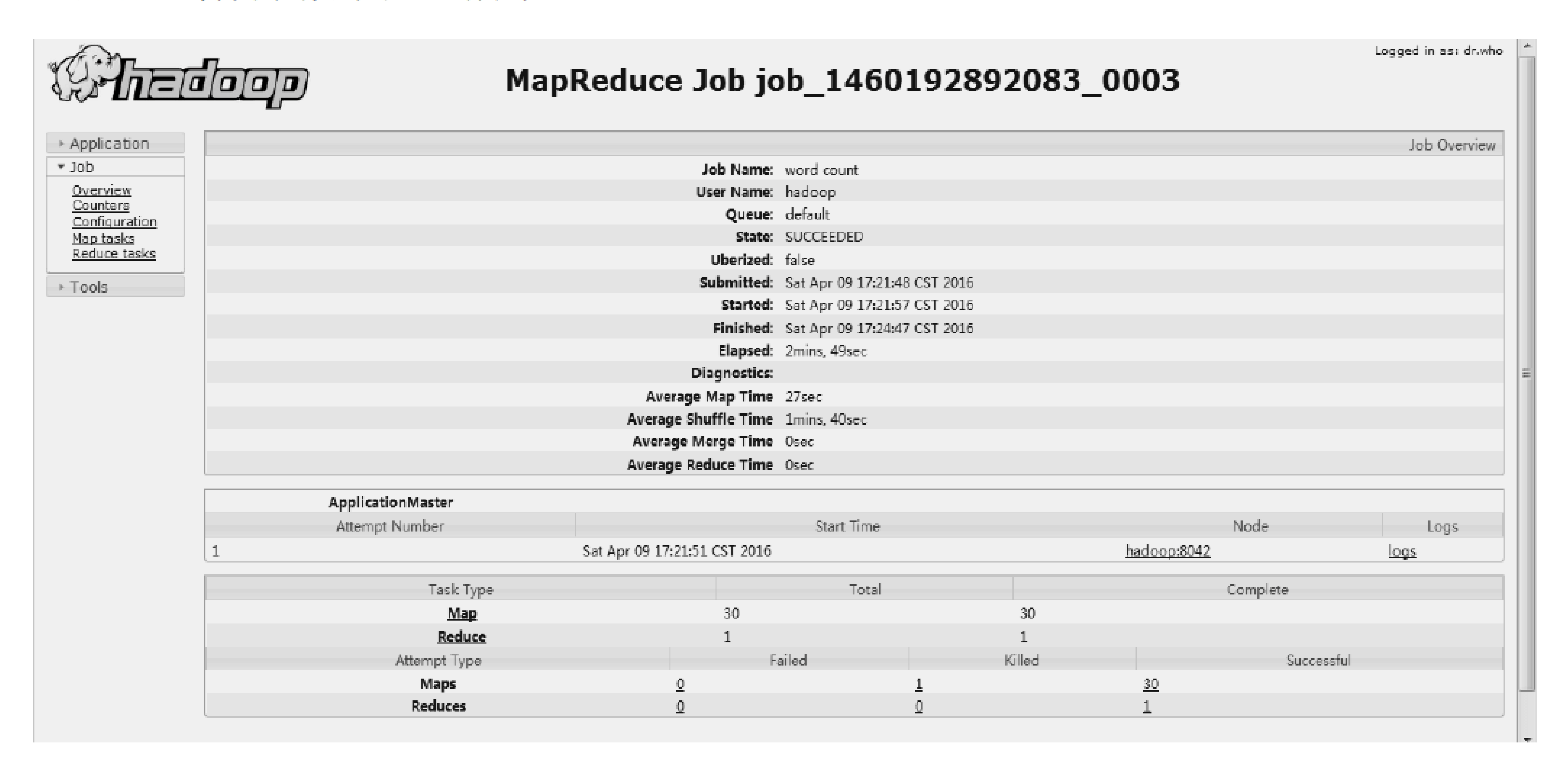


图 5-3　日志效果

点击 Map 或 Reduce 进入详细 task 界面，效果如图 5-4 所示。

hadoop　**Map Tasks for job_1460192892083_0003**

Application / Job: Overview, Counters, Configuration, Map tasks, Reduce tasks / Tools

Show 20 entries　Sear

Task					Successful Attempt	
Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Finish Time
task_1460192892083_0003_m_000000	SUCCEEDED	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016	31sec	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016
task_1460192892083_0003_m_000001	SUCCEEDED	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016	31sec	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016
task_1460192892083_0003_m_000002	SUCCEEDED	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016	31sec	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016
task_1460192892083_0003_m_000003	SUCCEEDED	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016	31sec	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016
task_1460192892083_0003_m_000004	SUCCEEDED	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016	31sec	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016
task_1460192892083_0003_m_000005	SUCCEEDED	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016	31sec	Sat Apr 9 17:22:00 +0800 2016	Sat Apr 9 17:22:31 +0800 2016
task_1460192892083_0003_m_000006	SUCCEEDED	Sat Apr 9 17:22:33 +0800 2016	Sat Apr 9 17:23:04 +0800 2016	30sec	Sat Apr 9 17:22:33 +0800 2016	Sat Apr 9 17:23:04 +0800 2016
task_1460192892083_0003_m_000007	SUCCEEDED	Sat Apr 9 17:22:33 +0800 2016	Sat Apr 9 17:23:04 +0800 2016	30sec	Sat Apr 9 17:22:33 +0800 2016	Sat Apr 9 17:23:04 +0800 2016
task_1460192892083_0003_m_000008	SUCCEEDED	Sat Apr 9 17:22:34 +0800 2016	Sat Apr 9 17:23:04 +0800 2016	30sec	Sat Apr 9 17:22:34 +0800 2016	Sat Apr 9 17:23:04 +0800 2016
task_1460192892083_0003_m_000009	SUCCEEDED	Sat Apr 9 17:22:34 +0800 2016	Sat Apr 9 17:23:04 +0800 2016	30sec	Sat Apr 9 17:22:34 +0800 2016	Sat Apr 9 17:23:04 +0800 2016
task_1460192892083_0003_m_000010	SUCCEEDED	Sat Apr 9 17:22:34	Sat Apr 9 17:23:04	30sec	Sat Apr 9 17:22:34	Sat Apr 9 17:23:04

图 5-4　任务的效果

选择一个 task 进入，细化信息如图 5-5 所示。

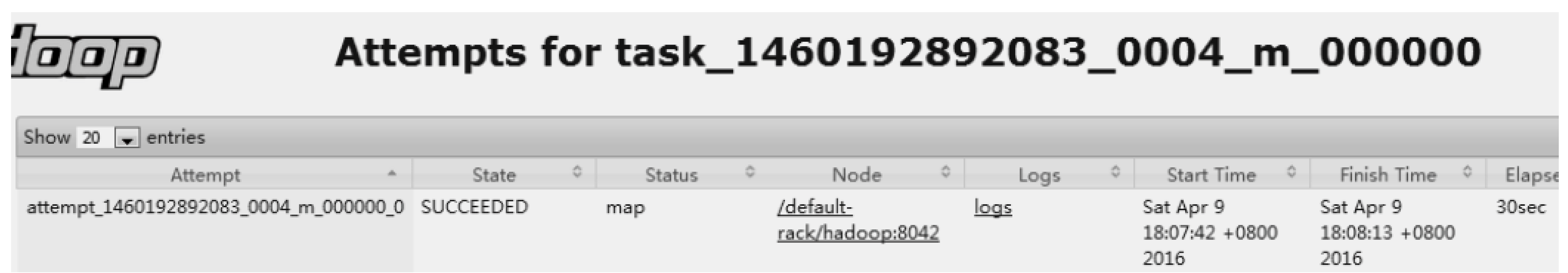


图 5-5　单任务的效果

点击 logs 即可查看本 task 的 stderr stdout 日志。

5.8　小　　结

本章介绍了 MapReduce 的工作原理、参数配置、YARN 集群搭建、MR 作业提交等常见操作。读者掌握了这些基本的 MR 操作、参数的作业及优化等相关内容，便可进行更高难度的 MR 程序实现。当然，Hadoop 集群、工作原理等理论知识对于更高效地使用 MR 提供了必要的理论基础。

非关系型数据库篇

第 6 章

使用 HBase

学习目标

HBase 是 Hadoop 平台中重要的非关系型数据库，它通过线性可扩展部署，可以支撑 PB 级数据存储与处理能力。本章将详细介绍 HBase 的基础组成与体系结构、安装配置、操作实践、优化管理等。

本章的目标是让读者能够初步掌握 HBase 这一非关系型数据库平台，为未来工作中进一步更好地使用 HBase 奠定基础。

本章要点

- HBase 的基本原理
- HBase 的组成模块
- HBase 的安装和配置
- HBase 的命令操作

6.1 HBase 基础

HBase 是一种非关系型数据库，具备单张表数百万列、数十亿行的数据管理能力，是实施大数据项目的重要利器之一。

6.1.1 HBase 是什么

HBase 的作用，是为大数据项目提供一种可伸缩、分布式、能够通过廉价机器进行线性扩展的数据库系统。相较于传统数据库的关系型特性，HBase 是一种基于列存储的非关系型数据库，这一特性确保了 HBase 能够满足更高并发的实时读写。

HBase 的诞生来源于 Google 的 *Bigtable: A Distributed Storage System for Structured Data* 论文，在论文中，Google 设计了一款名为 Bigtable 的数据库存储系统，用于解决它所面临的问题：如何为整个互联网用户提供实时的搜索查询结果？

这一问题可以细分成 4 个子问题：

- 如何存储抓取的整个互联网页面信息？
- 如何快速地通过用户关键词定位到这些信息？
- 如何实时可靠地响应数亿用户的并发请求？
- 在提升用户体验的同时，如何降低成本？

通过细分，可以看出，这些子问题其实是所有大数据服务公司在实际工作中所面临的共性问题。即：如何存储好自身不断增长的海量数据，如何更快、更可靠、更节省、更准确地将数据服务于用户。

借鉴 Bigtable 的方法，HBase 较好地解决了这一共性问题。

(1) HBase 可以运行于 Hadoop 的 HDFS 之上，所有的数据存储工作交由 HDFS 来完成，这确保了大规模的数据存储以及可扩展性。

(2) HBase 是面向列存储的非关系型数据库，基于<key，value>的查询机制，能够更快地定位要使用的数据。

(3) HBase 支持数百个以上节点的集群部署，并且在业务量高发期时，可通过增加节点提高性能，能够满足高并发访问需求。

(4) HBase 的硬件运行于廉价机器，软件本身开源，能够极大地节省实施成本。

由于上述特点，自 HBase 问世之后，便获得了业内广泛的关注与应用，大部分企业在自己的生产业务场景中直接部署 HBase，部分更有实力的企业借鉴于 HBase，改进甚至重写，打造自己的面向云的数据库系统，满足更苛刻、更个性化的业务场景。

HBase 自 2007 年创立起来，经过快速发展，目前已成为 Apache 软件基金会下面的顶级项目。正式的称呼是 Apache HBase。

HBase 的 Logo 是一个海豚头像，如图 6-1 所示。

HBase 官方网址是 http://hbase.apache.org/，在官网上可下载最新使用手册、最新版本、以及源代码。熟悉 Github 的读者，也可以通过 git://git.apache.org/hbase.git 链接下载和使用 HBase。这里需要提醒的是，Github 上面只是镜像。

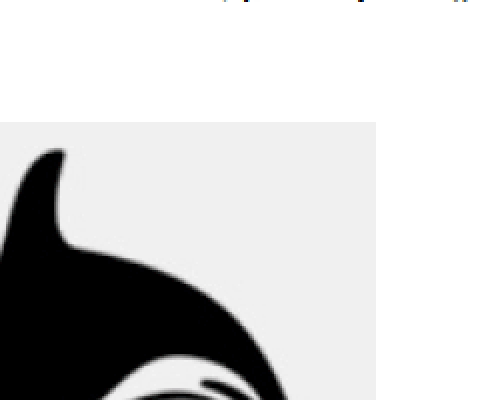


图 6-1　HBase 的图标

在 2015 年 2 月 HBase 正式发布了用于生产环境的 1.0 版本，这对 HBase 社区来说是一件大事。

不过，目前在生产环境中使用比较多的还是 HBase 0.94、0.98 版本，考虑到前面章节所采用的 HDFS 的版本，为确保稳定性与兼容性，本书所采用的 HBase 版本是 0.98。

如果读者想使用更新版本的 HBase，表 6-1 给出了所必需的 Hadoop 环境。

表 6-1　HBase 各版本所需的 Hadoop 环境

	HBase 0.94	HBase 0.98	HBase 1.0.x	HBase 1.1.x	HBase 1.2.x
Hadoop 1.0.x	不支持	不支持	不支持	不支持	不支持
Hadoop 1.1.x	支持	未验证	不支持	不支持	不支持
Hadoop 0.23.x	支持	不支持	不支持	不支持	不支持
Hadoop 2.0.x-alpha	未验证	不支持	不支持	不支持	不支持
Hadoop 2.1.0-beta	未验证	不支持	不支持	不支持	不支持
Hadoop 2.2.0	未验证	支持	未验证	未验证	未验证
Hadoop 2.3.x	未验证	支持	未验证	未验证	未验证
Hadoop 2.4.x	未验证	支持	支持	支持	支持
Hadoop 2.5.x	未验证	支持	支持	支持	支持
Hadoop 2.6.0	不支持	不支持	不支持	不支持	不支持
Hadoop 2.6.1+	未验证	未验证	未验证	未验证	支持
Hadoop 2.7.0	不支持	不支持	不支持	不支持	不支持
Hadoop 2.7.1+	未验证	未验证	未验证	未验证	支持

小提示

HBase 与 HDFS 的关系如下。

① HBase 并非绝对依赖于 Hadoop 才可以使用。

② 在伪分布式部署的教学或演示环境中，HBase 可以采用 Linux 本地文件系统，用于数据存储，也同样能够驱动 HBase 的运行。

③ 采用 HDFS 运行 HBase，是为了满足真正的生产环境的业务需求所追求的高稳定性、可靠性、扩展性。

6.1.2 HBase 伪分布式部署

HBase 与 HDFS 放置于同一台虚拟主机，主机名为 iZ23d4by1laZ，读者在自行配置时应注意替换。本次 HBase 采用伪分布式的方式进行部署，因此暂不需要 HDFS 等组件。

(1) HBase 的下载。

本书使用的版本是 HBase 0.98.3，而 HBase 官网最新的 0.98 版本是 HBase 0.98.16，两者属于同一版本系列的不同小版本，使用起来并无大的差异。

读者如果采用本书版本，可通过在本书下载资源中找到 hbase-0.98.3-hadoop1-bin.tar.gz，复制到相应的安装目录中即可。

如果要使用最新版本或者其他系列的 HBase 版本，可以通过以下网址进行下载：

```
http://apache.opencas.org/hbase/0.98.16.1/
```

在 Linux 中通过 wget 命令进行下载：

```
[root@www home]# wget
http://apache.opencas.org/hbase/0.98.16.1/hbase-0.98.16.1-hadoop1-bin.
tar.gz
--2016-01-12 15:35:18--
http://apache.opencas.org/hbase/0.98.16.1/hbase-0.98.16.1-hadoop1-bin.
tar.gz
Resolving apache.opencas.org... 159.226.11.160,
2001:cc0:2004:1:225:90ff:fe00:fe8b
Connecting to apache.opencas.org|159.226.11.160|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 75805159 (72M) [application/x-gzip]
Saving to: `hbase-0.98.16.1-hadoop1-bin.tar.gz'
```

(2) 解压缩。

将本书下载资源中的 hbase-0.98.3-hadoop1-bin.tar.gz 文件上传至/home/hadoop 目录中，然后执行解压缩与修改目录名操作。

具体的操作命令如下：

```
[root@www hadoop]# tar zxvf hbase-0.98.3-hadoop1-bin.tar.gz
[root@www hadoop]# ll
total 65168
drwxr-xr-x 7 root root     4096 Jan 12 15:41 hbase-0.98.3-hadoop1
-rw-r--r-- 1 root root 66654349 Jun 12  2014 hbase-0.98.3-hadoop1-bin.tar.gz
[root@www hadoop]# mv hbase-0.98.3-hadoop1 hbase
[root@www hadoop]# cd hbase
[root@www hbase]# ll
total 172
drwxr-xr-x  4 root root   4096 Jun  1  2014 bin
-rw-r--r--  1 root root 128252 Jun  1  2014 CHANGES.txt
drwxr-xr-x  2 root root   4096 Jun  1  2014 conf
drwxr-xr-x 28 root root   4096 Jun  1  2014 docs
drwxr-xr-x  7 root root   4096 Jun  1  2014 hbase-webapps
drwxr-xr-x  3 root root   4096 Jan 12 15:41 lib
-rw-r--r--  1 root root  11358 May 23  2014 LICENSE.txt
-rw-r--r--  1 root root    897 May 23  2014 NOTICE.txt
-rw-r--r--  1 root root   1377 Jun  1  2014 README.txt
[root@www hbase]#
```

(3) 修改配置文件。

切换到 HBase 的 conf 目录中，修改 hbase-env.sh、hbase-site.xml、regionservers 三个配置文件。

这三个文件的作用与修改效果操作如下。

① hbase-env.sh 文件：用于配置 HBase 环境变量，需要修改 JAVA_HOME、HBASE_MANAGES_ZK 两个参数，前者是配置 JDK 的版本，这里是 jdk1.8；后者的作用是 HBase 启动与关闭时是否同步启动与关闭 Zookeeper，为了方便管理，要设置为 true，即同步启动。

具体的配置信息如下：

```
[root@www conf]# vi hbase-env.sh
...
export JAVA_HOME=/usr/java/jdk1.8.0_05/
export HBASE_MANAGES_ZK=true
...
```

② hbase-site.xml 文件：用于配置 HBase 运行参数，重点是 hbase.master、hbase.rootdir 两个参数，前者是配置主 master 绑定的机器名与端口号；后者是配置 HBase 根数据放置的位置，本次部署是基于伪分布式，因此填充的值是本地目录(在使用前先创建好相应的目录)，如果要使用 HDFS，则此处要填充 HDFS 的文件系统路径。

具体的配置信息如下：

```
[root@www conf]# vi hbase-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
   <property>
      <name>hbase.master</name>
      <value>iZ23d4by1laZ:60000</value>
   </property>
   <property>
      <name>hbase.master.maxclockskew</name>
      <value>180000</value>
   </property>
   <property>
      <name>hbase.rootdir</name>
      <value>file:///home/hbase/data</value>
   </property>
   <property>
      <name>hbase.Zookeeper.quorum</name>
      <value>iZ23d4by1laZ</value>
   </property>
</configuration>
```

③ regionservers 文件：用于配置 HBase 的 regionserver 机器列表，这里只填充本地的主机即可。

具体的配置信息如下：

```
[root@www conf]# vi regionservers
iZ23d4by1laZ
```

6.1.3 服务的启动与验证

切换至/home/hbase/bin 目录，执行./start-hbase.sh 命令，用于启动 HBase 服务。通过 jps 命令查看三个进程 HMaster、HRegionServer、HQuorumPeer 是否启动成功；通过 netstat 命令查看三个服务所属的几个端口 60000、60010、60020、60030、2181 是否绑定成功，如果出现错误，读者可以通过查询 log 文件去跟踪出错的原因。

这里提到的几个端口的作用如表 6-2 所示。

表 6-2 HBase 主要端口的作用

端 口 号	隶属服务	作 用	配 置 项
60000	HMaster	用于 HBase 主节点的 RPC 服务	hbase.master 或 hbase.master.port
60010	HMaster	用于 HBase 主节点的 Web 服务	hbase.master.info.port
60020	HRegionServer	用于 HRegionServer 节点的 RPC 服务	hbase.regionserver.port
60030	HRegionServer	用于 HRegionServer 节点的 Web 服务	hbase.regionserver.info.port
2181	HQuorumPeer	用于 Zookeeper 守护服务	hbase.Zookeeper.property.clientPort

具体的启动与查看操作命令如下：

```
[root@www bin]# ./start-hbase.sh
...
[root@www bin]# jps
30528 Jps
10859 HMaster
9709 HQuorumPeer
8207 HRegionServer
[root@www bin]# netstat -nat
...
Proto Recv-Q Send-Q Local Address          Foreign Address          State
tcp        0      0  121.40.126.204:60000    0.0.0.0:*              LISTEN
tcp        0      0  0.0.0.0:2181            0.0.0.0:*              LISTEN
tcp        0      0  0.0.0.0:60010           0.0.0.0:*              LISTEN
tcp        0      0  121.40.126.204:60020    0.0.0.0:*              LISTEN
tcp        0      0  0.0.0.0:60030           0.0.0.0:*              LISTEN
...
```

6.1.4 HBase Shell 测试

在 bin 目录中执行./hbase shell 启动交互命令行，通过状态查询，创建一张测试表，添加与查询数据，停用并删除测试表，以测试部署的 HBase 是否可用。

(1) 启动 Shell。

在 bin 目录中有一个 hbase 命令，这个命令可用于管理 HBase 集群，输入./hbase 命令，可以查看能使用的管理命令：

```
[root@www bin]# ./hbase
Usage: hbase [<options>] <command> [<args>]
```

```
Options:
  --config DIR    Configuration direction to use. Default: ./conf
  --hosts HOSTS   Override the list in 'regionservers' file

Commands:
Some commands take arguments. Pass no args or -h for usage.
  shell           Run the HBase shell
  hbck            Run the HBase 'fsck' tool
  hlog            Write-ahead-log analyzer
  hfile           Store file analyzer
  zkcli           Run the Zookeeper shell
  upgrade         Upgrade HBase
  master          Run an HBase HMaster node
  regionserver    Run an HBase HRegionServer node
  Zookeeper       Run a Zookeeper server
  rest            Run an HBase REST server
  thrift          Run the HBase Thrift server
  thrift2         Run the HBase Thrift2 server
  clean           Run the HBase clean up script
  classpath       Dump HBase CLASSPATH
  mapredcp        Dump CLASSPATH entries required by MapReduce
  version         Print the version
  CLASSNAME       Run the class named CLASSNAME
```

执行./hbase shell，可以运行 HBase Shell 命令行，运行后，输入 status 可查询 HBase 集群服务的可用性状态：

```
[root@www bin]# ./hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.3-hadoop1, rd5e65a9144e315bb0a964e7730871af32f5018d5, Sat May
31 19:34:57 PDT 2014

hbase(main):001:0> status
...
1 servers, 0 dead, 3.0000 average load
...
```

(2)　创建测试表。

创建一张 testhbase 表，里面包括 test1、test2 两个字段：

```
hbase(main):002:0> create 'testhbase','test1','test2'
0 row(s) in 0.8370 seconds

=> Hbase::Table - testhbase
hbase(main):004:0> list
TABLE
testhbase
1 row(s) in 0.0130 seconds

=> ["testhbase"]
```

(3)　数据操作。

通过 put 命令向 testhbase 表中写入一条记录，填充 test1、test2 两个字段的内容；通过 get 命令查询写入的那条记录；通过 scan 命令检索 testhbase 的整张表：

```
hbase(main):002:0> put 'testhbase','1','test1','value1'
0 row(s) in 0.3840 seconds
```

```
hbase(main):003:0> put 'testhbase','1','test2','value2'
0 row(s) in 0.0120 seconds

hbase(main):004:0> get 'testhbase','1'
COLUMN                CELL
 test1:               timestamp=1453080535845, value=value1
 test2:               timestamp=1453080555303, value=value2
2 row(s) in 0.0550 seconds

hbase(main):005:0> scan 'testhbase'
ROW                   COLUMN+CELL
 1                    column=test1:, timestamp=1453080535845, value=value1
 1                    column=test2:, timestamp=1453080555303, value=value2
1 row(s) in 0.0520 seconds
```

(4) 删除测试表。

查看上一步创建的测试表，同时停用测试表，再将该表删除：

```
hbase(main):007:0> describe 'testhbase'
...
'testhbase', {NAME => 'test1', BLOOMFILTER => 'ROW', VERSIONS => '1',
IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'true false',
DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE',
MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536',
REPLICATION_SCOPE => '0'}, {NAME => 'test2', BLOOMFILTER => 'ROW', VERSIONS
=> '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'false',
DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE',
MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536',
REPLICATION_SCOPE => '0'}
1 row(s) in 0.3620 seconds

hbase(main):008:0> disable 'testhbase'
0 row(s) in 1.4070 seconds

hbase(main):009:0> drop 'testhbase'
0 row(s) in 0.1400 seconds

hbase(main):010:0> list
TABLE
0 row(s) in 0.0090 seconds

=> []
```

6.1.5 Web 测试

除了命令行外，HBase 提供了简单的基于 Web 的可视化管理手段，通过浏览器可以查看 HBase 的集群状态、配置状态、日志信息、表与数据信息等。HBase 中涉及两个重要节点，一是 Master 节点，用于 HBase 集群管理与调度；二是 RegionServer 节点，用于具体数据处理。

在 HBase 目录的 hbase-webapps 子目录中，存放着所有的 Web 管理页面，该目录分别为 master、regionserver、rest、static、thrift，分别用于存放不同功能的管理页面，其中，master 用于管理 Master 节点；regionserver 用于管理 RegionServer 节点；rest 用于对外的 rest 接口；thrift 用于对外的 thrift 接口；static 用于共同的 CSS、JS 等页面。

本小节重点演示 HMaster、HRegionserver 节点的 Web 管理界面。

(1)　Master 节点的 Web 管理。

在浏览器中输入“http://IP:60010”，对 IP 地址，读者要换成自己安装后的实际 IP 地址，可以打开并查看 HBase 中的 Master 节点的运行状态，如图 6-2 所示。

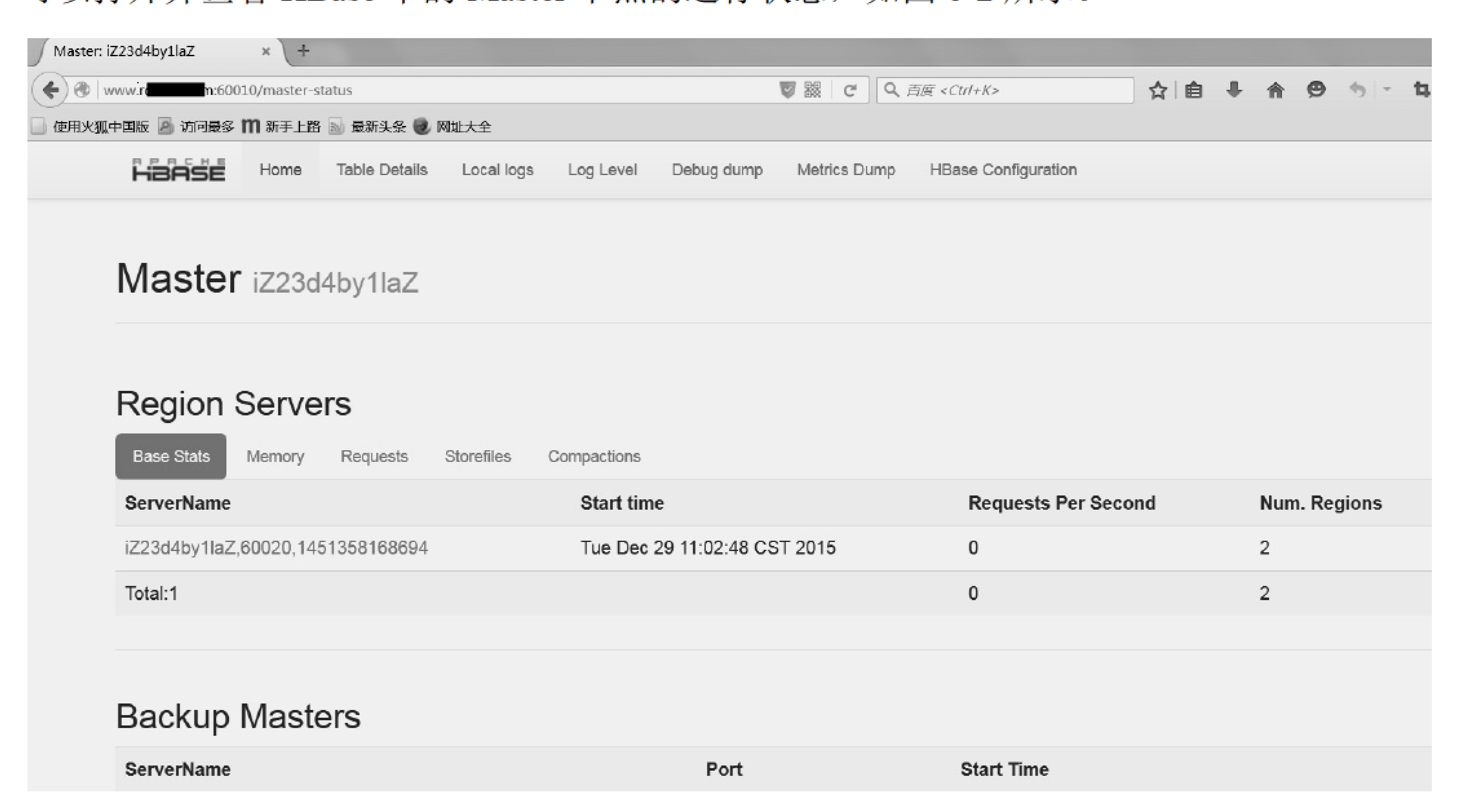


图 6-2　HBase Master 节点的 Web 管理

图 6-3 显示了当前 HBase 的表的细节信息，在上一小节创建了 testhbase 表，在图中可以看到该表的信息。

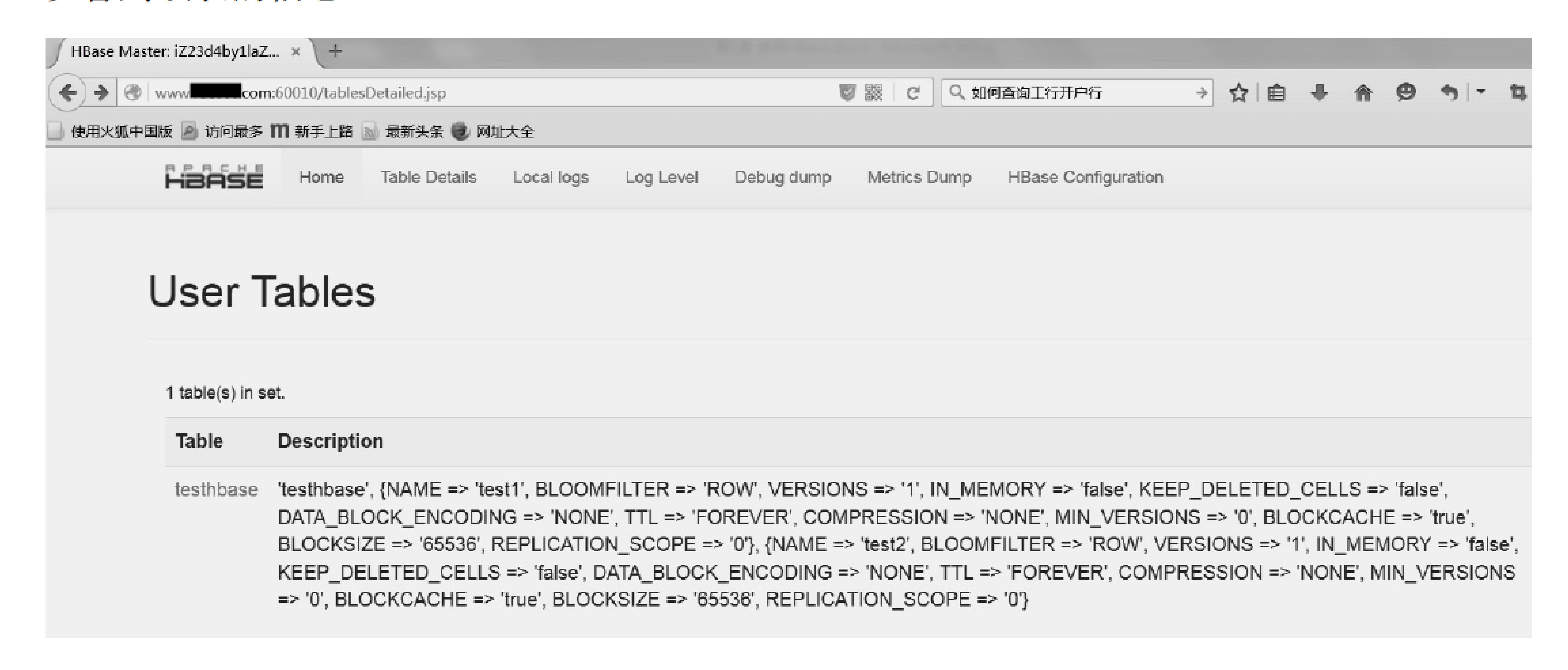


图 6-3　表信息的查看

(2)　RegionServer 节点的 Web 管理。

在浏览器中输入“http://IP:60030”，可以打开并查看 RegionServer 节点的运行状态，如图 6-4 所示。

RegionServer 用于具体的 HBase 数据与业务处理，通过 Web 界面能够实时监控服务的运行、资源消耗、队列等信息，如图 6-5 所示。

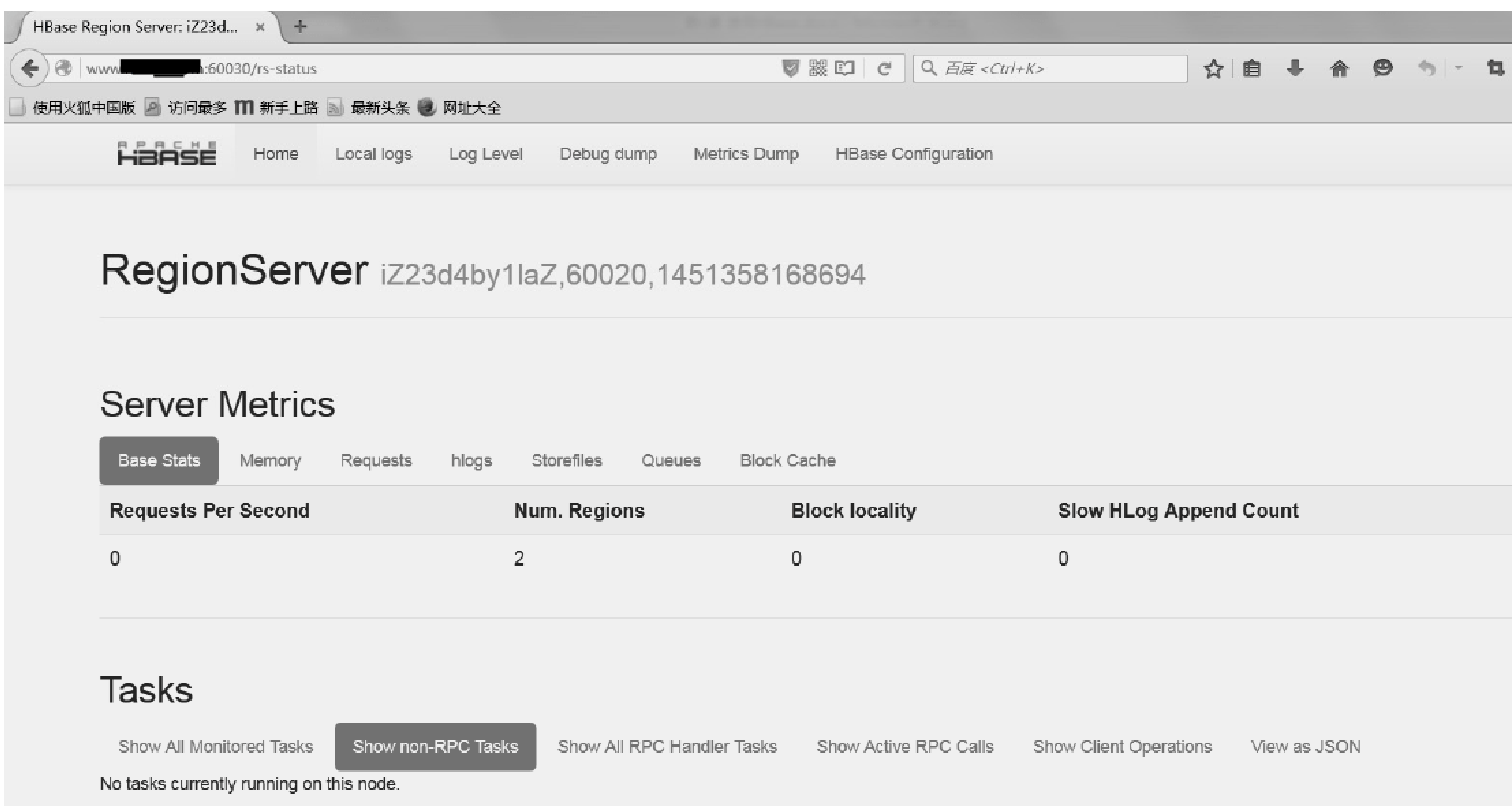


图 6-4　HBase RegionServer 节点的 Web 管理

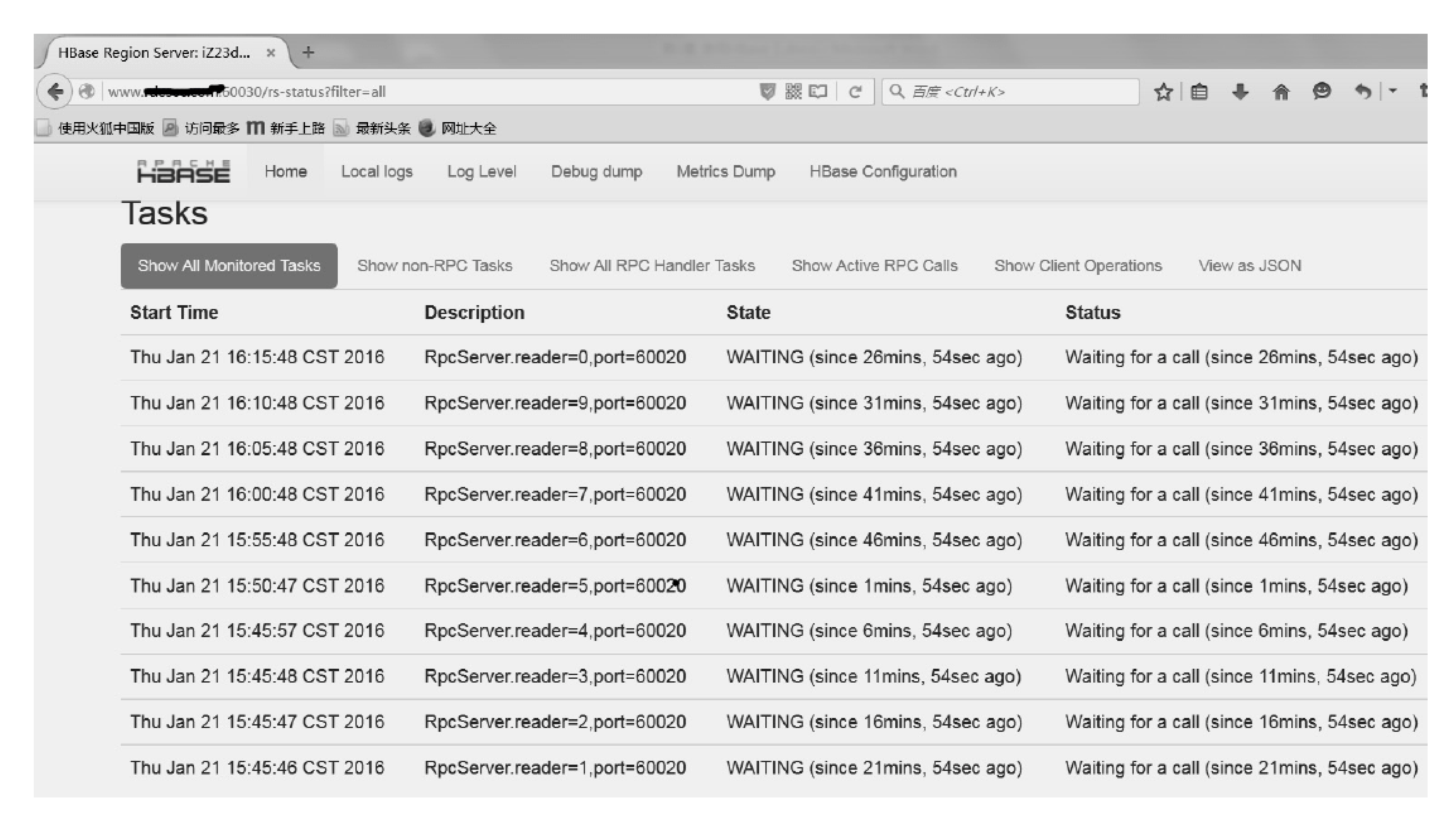


图 6-5　HBase RegionServer 的任务处理



小提示

关于 HBase 的 Web 管理。

① 目前，官方提供的版本主要定位于显示与监控。

② 有兴趣的读者可以尝试开发基于 Web 的 HBase 数据管理功能，如表、记录、列、数据导入、导出等。

6.1.6　服务的关闭

在 bin 目录中执行./stop-hbase.sh 可关闭 HBase，由于当前是伪分布式部署，所以只会关闭当前主机上的服务：

```
[root@www bin]# ./stop-hbase.sh
stopping hbase
...
[root@www bin]# jps
24463 Jps
```

小提示

关闭命令：在分布式集群中部署时，执行本命令，会关闭整个集群。

6.2　HBase 的架构原理

列存储机制、Table 与 Region 模型、体系架构、读写模型构成了 HBase 的基础，是实现其分布式特性的核心内容。

6.2.1　组成架构

HBase 由客户端、HMaster、HRegionServer、Zookeeper 组成。其中的客户端面向使用者；HMaster 负责 HBase 的全局事务调度；HRegionServer 负责 HBase 的具体数据存取，如果为伪分布式部署，则数据存放在本地；如果是分布式部署，则数据存放于 HDFS 集群；Zookeeper 用于实时感知 HBase 的各服务状态，保持服务与数据的一致性。

其整体架构效果如图 6-6 所示。

具体功能与描述如下。

(1)　客户端。

HBase 的客户端通过 RPC 的方式与 HBase 进行交互通信，基本原理是客户端向 HBase 发起连接，通过借助 Zookeeper 或直接访问 HMaster 去检索要访问的表数据具体位于哪一个 HRegionServer 节点，而后，向该 HRegionServer 节点发起具体的数据存取请求。如果访问管理层面的信息，则直接从 HMaster 处获取。

目前，HBase 提供了多种接口，来满足不同的用户访问需求，这些接口包括 HBase Shell、API、Thrift、REST、Hive 等。这些接口的描述分别如下。

①　HBase Shell：是 HBase 自带的命令行工具，能够实现 HBase 的基础管理功能。

②　API：HBase 提供了基于 Java 的 API 编程接口，这也是从事 HBase 开发最常用的接口使用方式，后面的内容会详细介绍如何基于 Java 开发基于 HBase 的应用，以及如何借助于 MapReduce 等服务处理海量数据。

③　Thrift：提供 Thrift 序列化技术，用于方便开发人员通过其他编程语言，如 PHP、C++、Python 等访问 HBase。

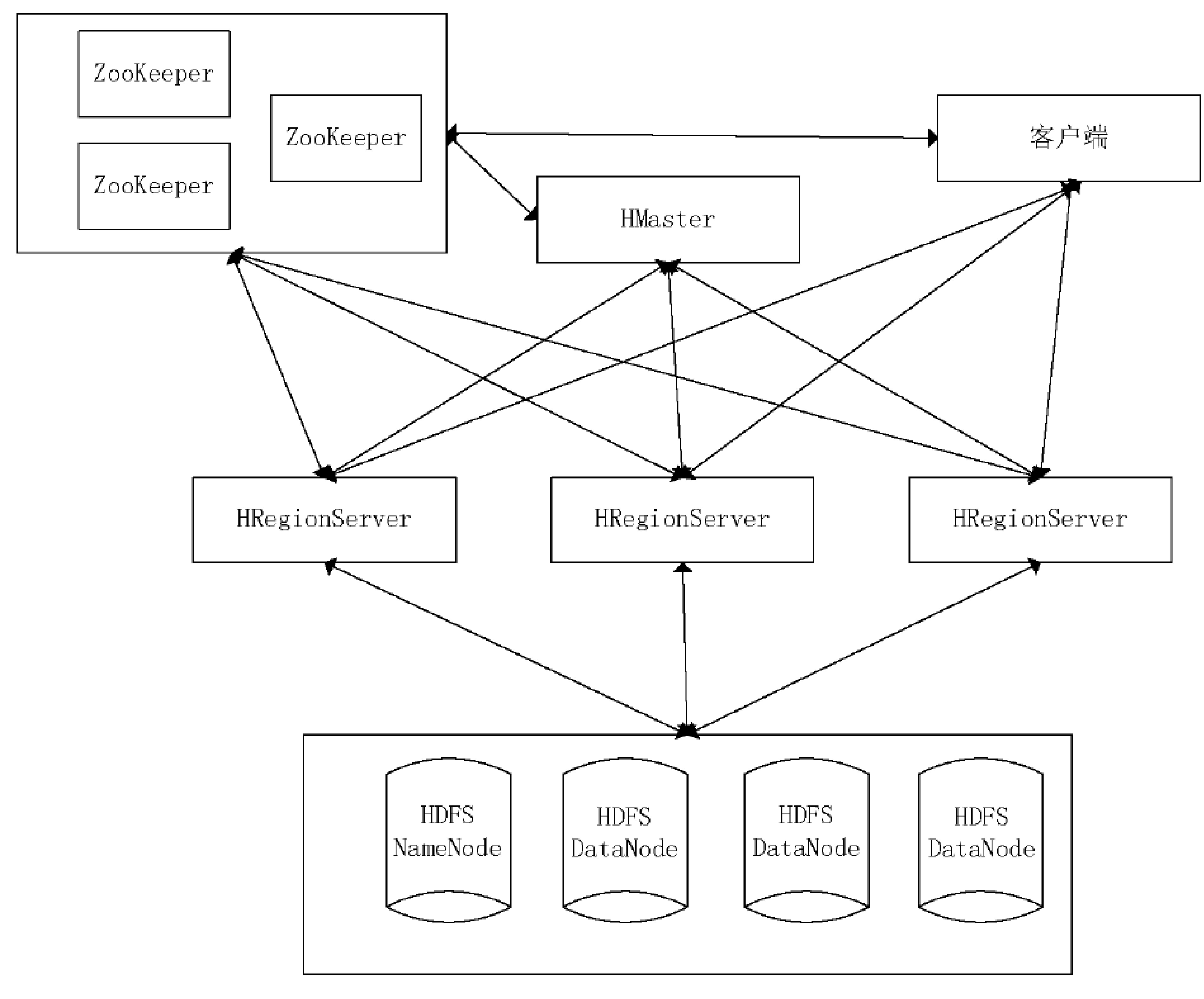


图 6-6　HBase 的组成架构

④　REST：基于 REST 的 HTTP API 的形式也可以去访问 HBase，这更大地降低了开发的难度。

⑤　Hive：借助 Hive 的 SQL 语法特性以及 MapReduce 处理能力，在 HBase 中，可以融合 Hive，借助 Hive 去处理 HBase 表数据。

小提示

关于客户端延伸。

①　读者可以基于熟悉的编程语言去开发 HBase 应用，通过开源社区，可以寻找适合项目的 HBase 管理工具。

②　有兴趣的读者可以开发一个基于 Web 的可视化 HBase 数据管理系统，类似于 MySQL 中的 PHPMyAdmin。

③　可以借助于 Hive 中的外部表，将表的数据源定向到 HBase，这样，可以实现两者之间的融合。

(2)　HMaster。

HMaster 可以理解为 HBase 的大脑，负责整个 HBase 的调度与管理，其主要作用如下。

①　负责元数据以及数据表的管理，包括表的增删改查、表的定义、表的变更、命名空间的定义等。

②　负责管理所有的 HRegionServer 节点，监测各节点的状态，负责各节点的上线、下

载，实现节点间的负载均衡。

③ 负责管理 HRegion 区域，每一个表由一个或多个 HRegion 区域构成，而每个 HRegionServer 负责管理一个或多个 HRegion 区域，HMaster 全局管理与分配哪一个 HRegion 应该放置于哪一台 HRegionServer 中；对于超出规模的 HRegion 区域进行分裂；负责停止服务的 HRegionServer 中的 HRegion 向其他节点的迁移等。

④ 负责全局安全策略管理。

⑤ 负责集群事务管理，如日志管理等。

⑥ 负责与 Zookeeper 集群交互。

在 HBase 的架构中，通常，HMaster 可以由单节点构成，但为了确保可靠性，也可以同时部署于两个以上节点，不过，多节点情形下，同时只能有一个节点处于主服务状态，其他节点要启动备用模式，处理备份的节点会定期与主节点进行信息同步，当主节点发生故障时，通过 Zookeeper 集群的选举机制，从备用节点中选出新的主服务节点。

小提示

关于 HMaster 与 HRegion。

① HMaster 部署位置也可与 HDFS 的名字节点位于同一台机器上。

② HMaster 负责维护-ROOT-、.meta 表，后面会介绍这两张表的作用。

③ HRegion 是 HBase 中基准的数据单元，后面会详细介绍其内容。

(3) HRegionServer。

HReginServer 可以理解为 HBase 的身体，负责 HBase 的数据存取工作，所有客户端对数据的最终读写操作都将落到 HRegionServer 上。一般，HRegionServer 的主要作用如下。

① 数据存储。借助本地文件系统或 HDFS 进行表数据、日志、缓存等数据的存储组织与管理。

② HRegion 管理。负责每个 HRegion 的状态维护、归并、迁移等工作。

③ WAL(HLog)管理。负责日志信息的管理，在 HBase 中，所有的数据更新需要先写入日志，再执行数据更新操作。

④ Metrics 管理。负责对外提供内部服务状况的参数，包括内存使用、Region 服务状况、Compaction、blockCache 等。由于 HBase Metrics 继承了 Hadoop Metrics，因此支持文件、数据流、Ganglia 等多种输出方式，便于外部监控。

⑤ 与客户端进行 RPC 交互，承担具体的数据读取与写入。

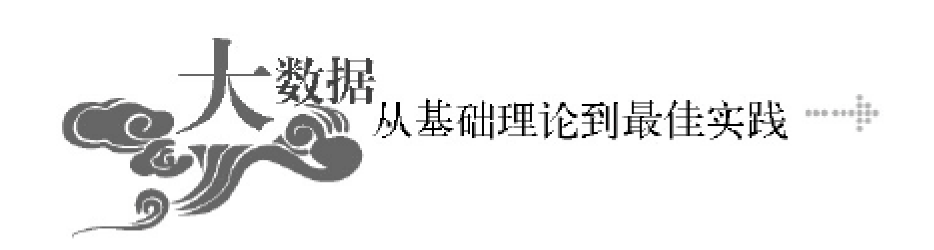
⑥ 与 HMaster 节点交互。查询元数据、上报自身数据状态，并按 HMaster 调度接管其他失效 HRegionServer 节点的数据与服务。

⑦ 与 Zookeeper 集群交互。确保分布式环境下的信息共享以及待执行任务的协同。

小提示

关于 HRegionServer。

① 在分布式环境中部署时，HRegionServer 一般和 HDFS 集群的 DataNode 在同一台



机器上。

② 每个 HRegionServer 节点包含多个 HRegion。

③ 每个 HRegionServer 节点可以同时承担多个表数据的存储。

(4) Zookeeper。

Zookeeper 是分布式应用程序协调服务，用于为分布式应用程序提供一致性服务，它是 Google Chubby 项目的开源实现，是 HBase 的重要支撑。

它在 HBase 中的主要作用如下。

① 负责存放 HBase 中的元数据与集群状态信息。

② 协调 HMaster 节点的主从切换，当检测到主 HMaster 节点宕机时，会通知备用 HMaster 节点进行接管，并知会所有 HRegionServer 节点。

③ 协调所有 HRegionServer 节点上线与下线，当检测到新的 HRegionServer 节点加入后，通知 HMaster 进行加入，管理集合；当检测到 HRegionServer 节点下线时，会通知 HMaster，并协同其他 HRegionServer 节点对宕机的 HRegionServer 节点的 HRegion 集合进行接管。

④ 面向客户端提供 RPC 服务端口。

在 HBase 中，提供了一个基于 Web 的 Zookeeper 状态查询页面，该页面比较简单，访问的地址为 http://ip:60010/zk.jsp，访问效果如图 6-7 所示。

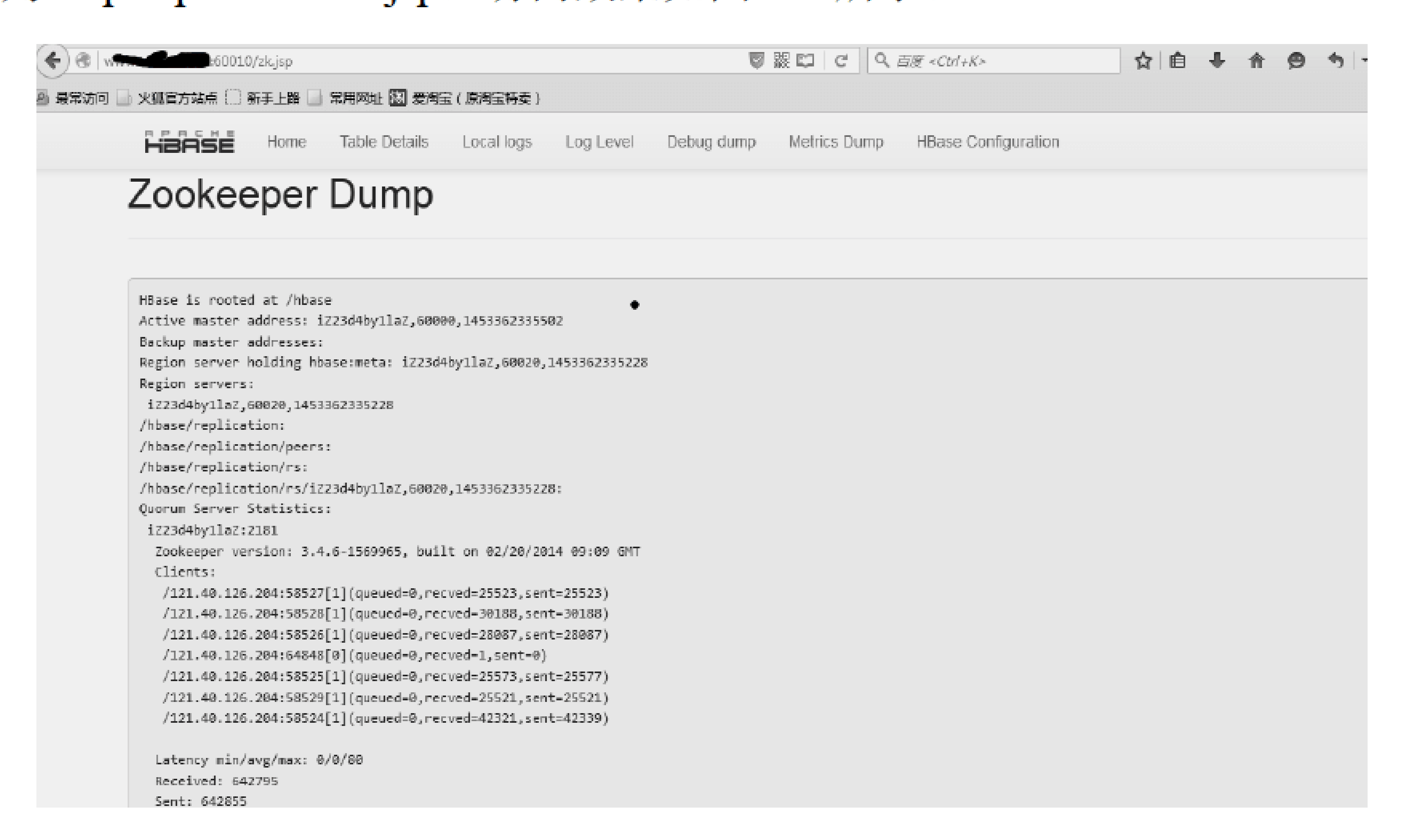


图 6-7 Zookeeper Dump 页面

小提示

关于 Zookeeper 集群。

① Zookeeper 可以独立部署在新的机器上，形成一套自主集群，也可以部署在 HMaster、HRegionServer 节点上。

② Zookeeper 集群部署要采用奇数个，这是由于其选举算法是当多个 Zookeeper 节点写成功时，任务数据才算成功，如果有 3 节点的 Zookeeper，其中 2 个节点写成功，即为成功；如果为 5 个节点，则其中 3 个节点写入成功，即为成功。

6.2.2　数据模型

HBase 由表(Table)、行(Row)、行键值(Row Key)、列簇(Column Family)、列(Column)、版本(Version)、时间戳(timestamp)等组成。其数据模型如图 6-8 所示。

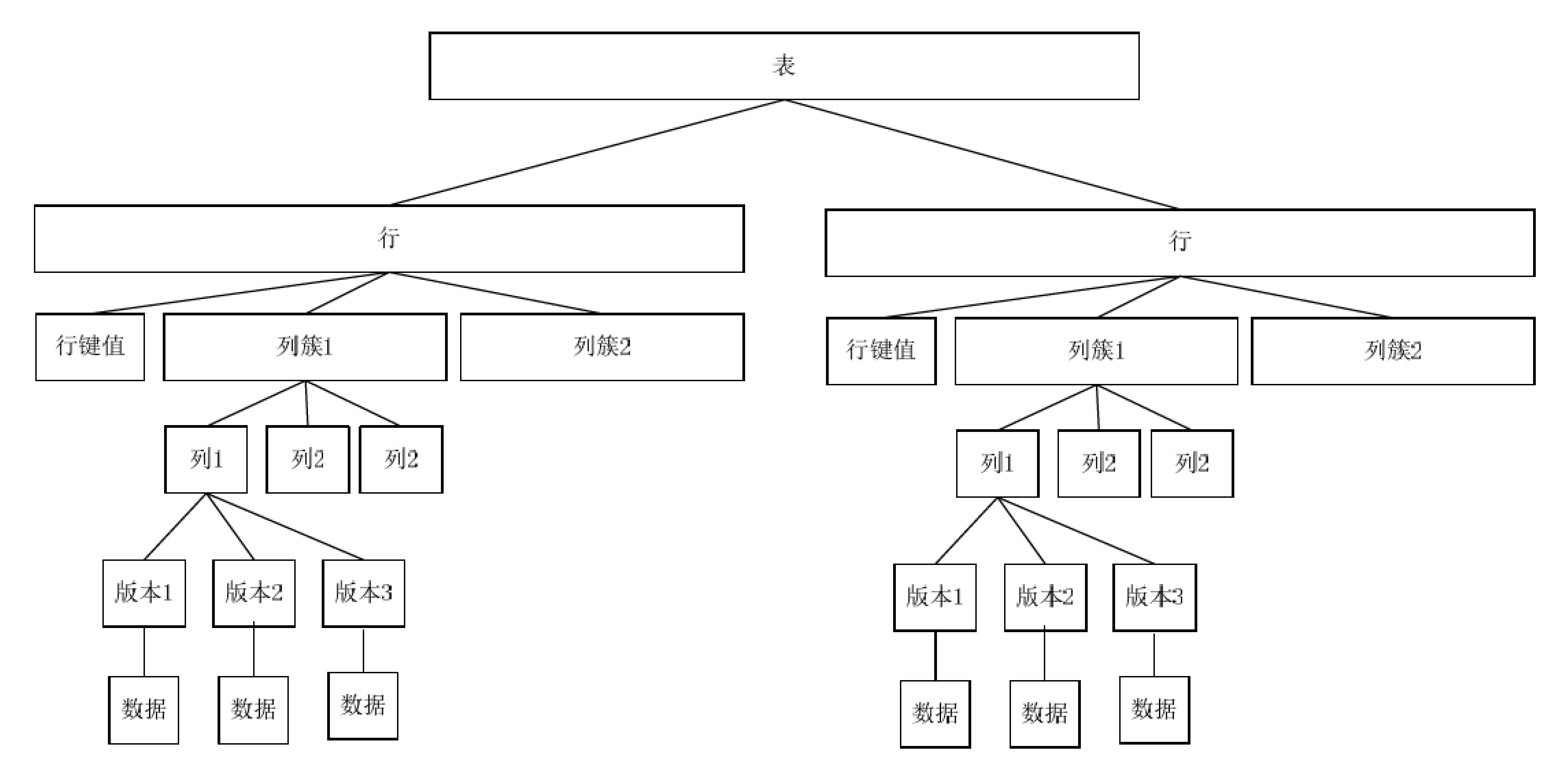


图 6-8　HBase 的数据模型

(1)　表(Table)。

表是 HBase 数据存储形式，HBase 可定义多张数据表，每张数据表由多行组成。

(2)　行与行键(Row & Row Key)。

传统关系型数据库基于行存储的，而 HBase 是基于列存储的，即<key，value>机制，每一行记录由行健、多个列簇构成，每个行键是一个字符串，以数组的形式存储，以英文字典序的形式排序，长度最大不能超过 64KB。

访问 HBase 表中的记录时，只能通过查询行键的方法去获取，在 HBase 中，提供了三种访问方法。

- 单行访问：通过查询单个行键值进行访问。
- 区间访问：通过设定行键值区间进行访问。
- 全表访问：通过对全表做扫描进行访问。

小提示

行键与性能。

①　通常，一个行键长度可控制在 100 字节以内，可提高查询性能。

②　将经常一起操作的行放在一起，通过位置相关性，可提高查询性能。

(3)　列簇(Column Family)与列(Column)。

在 HBase 中，数据部分可由多个列簇(Column Family)构成，每个列簇由一个或多个列组成，这与传统的关系型数据库的行、列的概念有了很大的区别。

例如，假定每个人的年收入(income)由工资收入(salary)、奖金收入(bonus)、利息收入(interest)构成，在传统关系型数据库里，如果要定义员工收入表，上述的三部分会构成一个表的三个列；而在 HBase 里，设计时可以定义一个收入列簇 income，上述三部分是 income 列簇里面的三个列，即工资收入 income:salary，奖金收入 income:bonus，利息收入 income:interest，从表述中可以看出，要操作 HBase 表中的某一列，需要先指定其列簇前缀，才可以访问。

HBase 的访问控制、数据存储是基于列簇这个基础的，列簇与列簇之间可能是分布在不同的物理机器上的，因此，要提高访问性能，需要将关联性比较高的列放在同一列簇里。此外，在同一列簇里的不同的行，可以存在不同的列，列簇里的列可以无须声明直接使用。

例如，公司在发展，员工逐渐划分出多个层级，有些层级的员工(如高管)开始享有股份激励收入(stock)，传统的关系数据库需要重新修改表结构，但在 HBase 中无须这么做，每一行代表一个员工，只需针对需要享受此项收入的员工所对应的行提交 income:stock 列及相关数据，即可完成此收入的填报，而无须定义此列。

小提示

关于列簇：通常，一个 HBase 表不要定义太多个列簇，否则会影响性能，比较好的数量是 2 到 3 个列簇。

(4) 版本(Version)、时间戳(Timestamp)与数据(data)。

HBase 在写入数据时的时间戳极为关键，它是精确到毫秒的一个 64 位整型数，可以是系统自动填充的，也可以由用户显式赋值。对一个具体的行、列，每次提交的数据与时间戳构成一次数据版本，每个行的每一列都可以存储多个数据版本，不同数据版本之间通过时间戳进行排序，最近的排在最前面，用户访问某一行某一列时，默认会返回最新提交的内容，而通过<行键、列、版本号>可以访问某一行、某一列指定时间的数据。

例如，针对员工张三，每年工资收入都会有变化，我们可以在 2014 提交一次 20000；2015 提交一次 25000；2016 提交一次 30000；当 2016 年访问 income:salary 字段时，返回的结果为 30000，如果我们想了解 2015 的收入水平，在 HBase 中极为简单，只需要访问时加上版本号，即可返回 2015 年的工资收入，如果是传统关系型数据库，则会相对复杂，需要额外建立一个工资收入的历史信息表。

小提示

关于版本。

① 每一列可以定义多个版本，通过设定参数 HColumnDescriptor 可以定义每个列簇的最大版本数、最小版本数。

② 版本回收可以通过设定版本总数或者设定某一时期内的版本总数进行控制。

(5) 示例：HBase 数据模型。

结合本小节所提到的员工收入示例，表 6-3 给出了在 HBase 中的逻辑数据模型。

表 6-3　HBase 中的员工收入表

StaffIncome 表					
行键值 (Row Key) (员工 ID)	时间戳 (TimeStamp)	income 列簇(column family)			
		salary	bonus	interest	stock
A	T7	30000			12000
	T3	25000	3000	1000	10000
	T1	20000	2000		
B	T6	10000			
	T2	6000	3000		
C	T5	8000	1000	2000	

针对上述数据表，有以下解读。

① 数据表类似于一张二维的稀疏矩阵，实际上，HBase 中的数据表本质上就是一个稀疏矩阵。

② 数据表中的数据更新可以一次更新本行中所有的列，也可以只更新本行中某一列或某几列，此外，数据表可以灵活地在列簇中扩展数据列，无须定义。

③ 数据表的定义以员工 ID 为行键值，通过查找员工 ID，能够定位到具体的记录，默认查询时，会返回最新时间戳的记录，如查询 A 员工所有收入，返回的值为<salary:30000，bonus:3000，interest:1000，stock:12000>，B 员工的所有收入返回值为<salary:10000，bonus:3000>，员工 C 的所有收入返回值为<salary:10000，bonus:1000，interest:2000>。

④ 通过时间戳，可以直观地看到某一行、某一列的历史版本信息，设定版本号范围，即可以查询先前的数据情况，这种机制本身，能够缓解因数据更新误操作而带来的数据安全问题。

6.2.3　物理存储

传统的关系型数据库管理系统会随着数据表规模的不断扩大，面临容量与性能两方面的问题，在 HBase 表的物理存储中，采用了区域的概念，即 HBase 中的每张表默认只有一个区域(Region，用于保存数据表中某段连续的数据，如存储数据表中的若干行)，随着数据量的增长，会被拆分成两个或多个区域，在物理存放时，Region 会被存储至不同的 HRegionServer 上，形成了分布式的存储模式。

在 HBase 中，一张表可以由多个存放到不同服务器上的 Region 构成，HBase 表、HRegion、HRegionServer 之间关系可由图 6-9 来示意。

将每张表按行键值区域划分出不同的 Region，每个 Region 分布式存储到不同的 RegionServer 服务器上。对于用户的访问请求，根据所输入的行键值，定位到所访问的 Region 存储的具体 HRegionServer，而后通过与该 HRegionServer 的交互，获取具体的数据信息。

基于这种分布式的存储访问机制，一方面解决了表数据容量的线性扩展问题，如果表数据量激增(TB 乃至 PB 级)通过加入相对应数量的 HRegionServer 服务器，即可满足容量要

求；另一方面，由于具体的数据交互是与 HRegionServer 进行的，而每个 HRegionServer 只存储一定数量的 Region，因此，增加 HRegionServer 就可以提升集群整体并发访问能力。

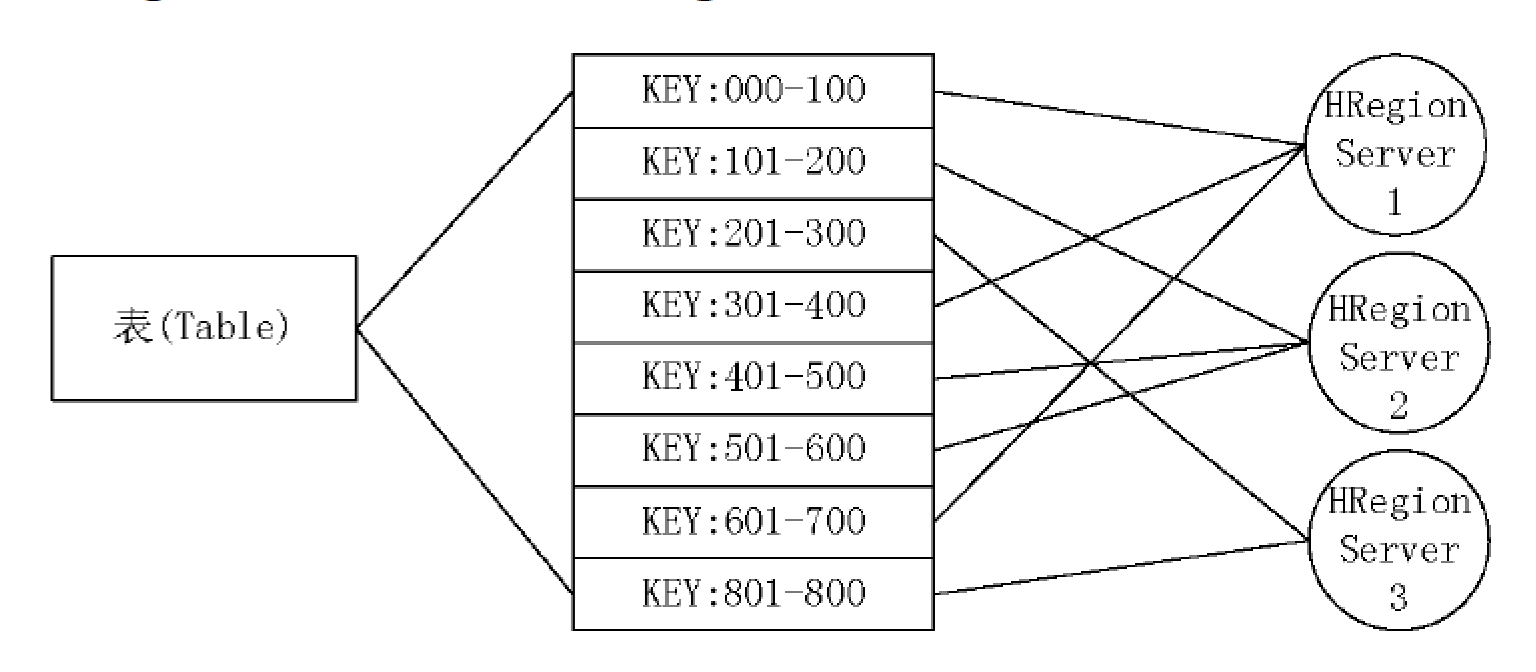


图 6-9　表存储模型

(1)　特殊的表。

在 HBase 中有两个特殊的表，一个是-ROOT-；另一个是.meta，这两张表的作用，是支撑 HBase 中的数据寻址，其中，-ROOT-表存储.meta 表中的 Region 位置信息；.meta 表存储 HBase 中所有用户数据表的 Region 位置信息。

检索过程的效果如图 6-10 所示。

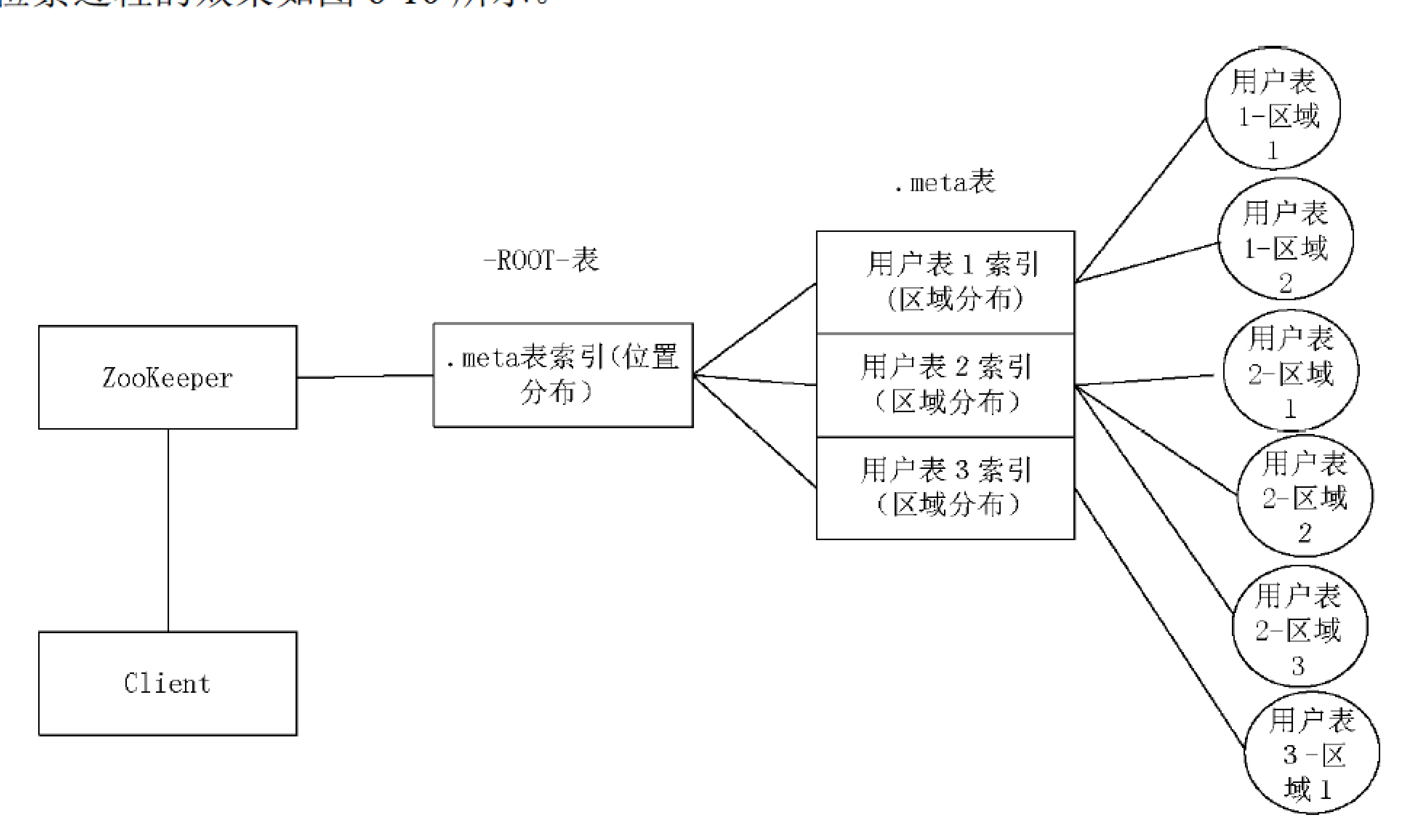


图 6-10　数据访问检索模型

-ROOT-的存放位置由 Zookeeper 记录，对于用户的访问请求，首先查询-ROOT-表，查看要访问的数据表存放在哪个.meta 表中，然后去查找相对应的.meta 表，获取目标数据表的区域分布情况，通过比对行键值，获取具体存放数据的 HRegionServer，最后去访问 HRegionServer，获操作数据。

HBase 面临着 PB 级数据表的存储压力，这些数据表的 Region 的数量也极为海量，因而存储这些 Region 索引信息的.meta 表也会变得非常大，在 HBase 管理时，会将.meta 表拆分成多个 Region，这些 Region 同样会存放至不同的 HRegionServer 上，对于用户的频繁数据访问，如果每次访问都需要直接查询.meta 表，则效率会非常差，因此，HBase 引入的

-ROOT-用于存储.meta 的 Region 分布信息，同时，为了提升性能，以及考虑到针对.meta 表的索引数据量不会特别大，-ROOT-表由一个 Region 组成。

-ROOT-、.meta 表的格式极为相似，行键值由 tableName、regionStartKey、regionId、replicaId 等信息组成，列簇由一个 info 组成，info 列簇由 regioninfo、server、serverstartcode 三列组成。其中，RegionInfo 列存放区域描述信息，由 regionId、tableName、startKey、endKey、offline、split、replicaId 等组成；server 列存放对应的 HRegionServer 服务器，包括信息 server 与 port；serverstartcode 对应时间信息，为 HRegionServer 的启动时间戳。

HBase 将.meta 表存放于命名空间 hbase 中，可以通过如下命令查看其元结构：

```
hbase(main):105:0> describe 'hbase:meta'
DESCRIPTION
ENABLED

'hbase:meta', {TABLE_ATTRIBUTES => {IS_META => 'true', coprocessor$1 =>
'|org.apache.hadoop.hbase.coprocessor.MultiRowMutationEndpoint|536870911
|'}, {NAME => 'info', BLOOMFILTER => 'NONE', VERSIONS => '10', IN_MEMORY =>
'true', KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL
=> 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true',
BLOCKSIZE => '8192', REPLICATION_SCOPE => '0'}
1 row(s) in 0.0240 seconds…
```

通过 scan 命令可以扫描 info 列簇中的各列信息，如下所示：

```
hbase(main):108:0> scan 'hbase:meta'
ROW                                 COLUMN+CELL
hbase:namespace,,1451358713278.2da column=info:seqnumDuringOpen,
timestamp=1453362348280, value=\x00\x00\x00\x00\x00\x00\x00\x05
d5136e3f76dab12bccd76c213121b.hbase:namespace,,1451358713278.2da
column=info:server,
timestamp=1453362348280,
value=iZ23d4by1laZ:60020d5136e3f76dab12bccd76c213121b.hbase:namespace,,
1451358713278.2da column=info:serverstartcode,
timestamp=1453362348280,
value=1453362335228d5136e3f76dab12bccd76c213121b.
...
testhbase,,1453080469131.0c3da9d15 column=info:regioninfo,
timestamp=1453080469185, value={ENCODED =>
0c3da9d154163bbb2b106ae878acc9cc,
4163bbb2b106ae878acc9cc. NAME =>
'testhbase,,1453080469131.0c3da9d154163bbb2b106ae878acc9cc.', STARTKEY
=>'', ENDKEY => ''}
testhbase,,1453080469131.0c3da9d15 column=info:seqnumDuringOpen,
timestamp=1453362348279, value=\x00\x00\x00\x00\x00\x00\x00\x05
4163bbb2b106ae878acc9cc.testhbase,,1453080469131.0c3da9d15
column=info:server,
timestamp=1453362348279, value=iZ23d4by1laZ:600204163bbb2b106ae878acc9cc.
testhbase,,1453080469131.0c3da9d15 column=info:serverstartcode,
timestamp=1453362348279, value=14533623352284163bbb2b106ae878acc9cc.
```

在 HBase 中还有一张 namespace 表，这张表的作用是存储 HBase 所用命名空间的元数据信息，默认时，HBase 会自带 hbase、default 两个命名空间，它的格式与列簇信息如下：

```
hbase(main):110:0> describe 'hbase:namespace'
DESCRIPTION
ENABLED
'hbase:namespace', {NAME => 'info', BLOOMFILTER => 'ROW', VERSIONS =>
'10',true
```

```
IN_MEMORY => 'true', KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING =>
'NONE', TTL => 'FOREVER ', COMPRESSION => 'NONE', MIN_VERSIONS => '0',
BLOCKCACHE => 'true', BLOCKSIZE => '8192 ', REPLICATION_SCOPE => '0'}
1 row(s) in 0.0280 seconds
hbase(main):111:0> scan 'hbase:namespace'
ROW                            COLUMN+CELL
default                        column=info:d, timestamp=1451358714056,
value=\x0A\x07default
hbase                           olumn=info:d, timestamp=1451358714080,
value=\x0A\x05hbase
2 row(s) in 0.0110 seconds
```

(2) HRegion。

整个 HBase 集群中，每台 HRegionServer 会存储多个 HRegion(H+Region 以区分为 HBase 的 Region)，这些 HRegion 可以来源于多个数据表。HRegion 作为基础的数据存储单元，由 HMemcache(缓存文件)、HLog(日志文件)、HStore(持久文件)三部分组成。HLog 存放日志信息，用于日志回滚与查看； HMemcache 存放缓存数据，用于数据更新以及数据查询；HStore 存放持久化数据，用于数据物理存储。

在 HBase 中，HRegionServer、HRegion、HMemcache、HLog、HStore 相互之间的关系如图 6-11 所示。

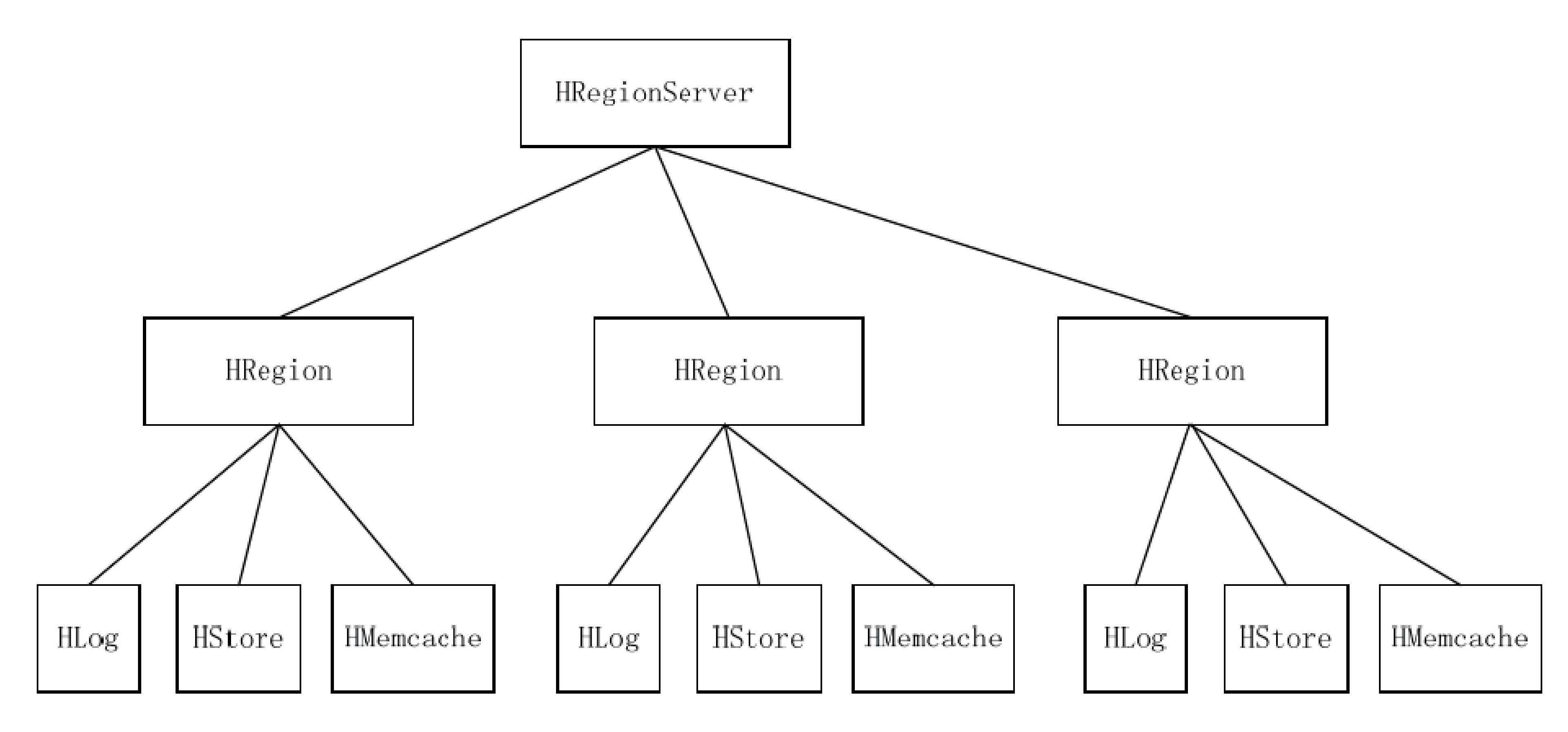


图 6-11 物理存储模型

对于一条要更新的数据，在写入时，先检查 HMemcache 中有无相同的更新请求，如有则返回；如果没有，则先在 HLog 中写入日志，而后写入至 HMemcache 中，HMemcache 在一定周期内，会将待写入的所有数据请求刷新至 HStore 中，而 HStore 在集群中与 HDFS 交互，负责数据的持久化写入。

6.3 HBase 的命令实践

HBase Shell 中涉及到命名空间、表、数据增删改查、工具、权限等命令，这些命令对于掌握并管理 HBase 是非常关键的。

6.3.1　概述

HBase 提供了九大类命令，包括通用(general)、DDL、DML、命名空间(namespace)、工具(tools)、快照(snapshot)、安全(security)、可视化标签(visibility labels)、复制(replication)等，每一类命令的具体作用如表 6-4 所示。这些命令可以帮助管理与操作 HBase 集群。

表 6-4　HBase 命令分类

分　类	作　用
通用	信息查看命令，如 HBase 版本、HBase 当前状态
DDL	数据定义语言，如表的定义、修改、删除
DML	数据操纵语言，如记录的新增、修改、删除、事务处理
命名空间	表的逻辑分组管理，如，一个命名空间可包括多张表
工具	HBase 工具，用于集群管理，如调整 Region 等
快照	HBase 快照，如删除快照、创建快照
安全	权限与安全管理，如分配权限、回收权限
可视化标签	HBase 用户或表信息的描述标签，如添加标签
复制	节点管理，如添加节点、删除节点、数据复制，使用前 hbase.replication 状态须置为 true

在 HBase Shell 中输入 help 命令，可以查看所有分类及其所属命令的清单。

```
hbase(main):005:0>help
...
COMMAND GROUPS:
  Group name: general
  Commands: status, table_help, version, whoami

  Group name: ddl
  Commands: alter, alter_async, alter_status, create, describe, disable,
disable_all, drop, drop_all, enable, enable_all, exists, get_table,
is_disabled, is_enabled, list, show_filters

  Group name: namespace
  Commands: alter_namespace, create_namespace, describe_namespace,
drop_namespace, list_namespace, list_namespace_tables

  Group name: dml
  Commands: append, count, delete, deleteall, get, get_counter, incr, put,
scan, truncate, truncate_preserve

  Group name: tools
  Commands: assign, balance_switch, balancer, catalogjanitor_enabled,
catalogjanitor_run, catalogjanitor_switch, close_region, compact, flush,
hlog_roll, major_compact, merge_region, move, split, trace, unassign,
zk_dump

  Group name: replication
  Commands: add_peer, disable_peer, enable_peer, list_peers,
list_replicated_tables, remove_peer, set_peer_tableCFs, show_peer_tableCFs

  Group name: snapshot
```

```
  Commands: clone_snapshot, delete_snapshot, list_snapshots,
rename_snapshot, restore_snapshot, snapshot

  Group name: security
  Commands: grant, revoke, user_permission

  Group name: visibility labels
  Commands: add_labels, clear_auths, get_auths, set_auths, set_visibility
...
```

对于某一分类，输入 help ‘groupname’，可查看该分类的描述，如查看 general 分类的使用：

```
hbase(main):032:0> help 'general'
Command: status
Show cluster status. Can be 'summary', 'simple', or 'detailed'. The
default is 'summary'. Examples:

  hbase> status
  hbase> status 'simple'
  hbase> status 'summary'
  hbase> status 'detailed'

Command: table_help
Help for table-reference commands.
...
```

对于某个命令，输入 help “command”，可查看其具体的使用语法，例如，查看 grant 命令的使用：

```
hbase(main):013:0> help "grant"
Grant users specific rights.
Syntax : grant <user> <permissions> [<table> [<column family> [<column
qualifier>]]

permissions is either zero or more letters from the set "RWXCA".
READ('R'), WRITE('W'), EXEC('X'), CREATE('C'), ADMIN('A')

For example:

    hbase> grant 'bobsmith', 'RWXCA'
    hbase> grant 'bobsmith', 'RW', 't1', 'f1', 'col1'
    hbase> grant 'bobsmith', 'RW', 'ns1:t1', 'f1', 'col1'
```

6.3.2 命名空间

从 HBase 0.98 版本开始支持命名空间(namespace)，命名空间用于对 HBase 中的表进行逻辑管理，类似于传统 DBMS 中数据库的概念，一个数据库里可以建多张表，同理，在 HBase 的命名空间里，可以管理多张表。

HBase 命名空间的命令包括：本身支持的创建、删除、查询等操作。

(1) 创建命名空间。

使用 create_namespace 命令，后面跟命名空间的名称，可以创建一个新的命名空间，这里创建一个名为 bigdata 的命名空间：

```
hbase(main):008:0> create_namespace 'bigdata'
0 row(s) in 0.0280 seconds
```

(2)　查看命名空间。

使用 list_namespace 命令，可以列出当前所有的命名空间。在 HBase 中，默认的命名空间有两个，一个是 hbase，是系统内建的，用于存放 meta、namespace 两张表；另一个是 default，对于没有指定命名空间创建的表，默认都会放置到 default 命名空间中。

```
hbase(main):009:0> list_namespace
NAMESPACE
bigdata
default
hbase
3 row(s) in 0.0680 seconds
```

list_namespace 命令后面可以跟命名空间名称，用于模糊查找符合正则表达式的命名空间。例如，创建一个新的命名空间 big，然后通过模糊查询去查看所有满足前缀为 big 的命名空间：

```
hbase(main):015:0> create_namespace 'big'
0 row(s) in 0.0270 seconds

hbase(main):017:0> list_namespace 'big*'
NAMESPACE
big
bigdata
2 row(s) in 0.0110 seconds
```

(3)　查看命名空间的表。

list_namespace_tables 命令列出指定命名空间所有的表的清单。例如，列出当前 hbase 命名空间中的所有表：

```
hbase(main):020:0> list_namespace_tables 'hbase'
TABLE
meta
namespace
2 row(s) in 0.0310 seconds
```

(4)　查看命名空间描述。

describe_namespace 命令用于查看指定命名空间的描述。例如，查看当前 hbase 命名空间的描述信息：

```
hbase(main):021:0> describe_namespace 'hbase'
DESCRIPTION
{NAME => 'hbase'}
1 row(s) in 0.0170 seconds
```

(5)　修改命名空间。

alter_namespace 命令用于修改命名空间属性，修改的方式可以是设定一个属性以及属性值，也可以是取消一个属性。

这里为 bigdata 设定两个属性信息，设定的规则为{METHOD=>‘set’, ‘key’=>‘value’}，在 key 与 value 中填充要设定的属性名和属性值：

```
hbase(main):024:0> alter_namespace 'bigdata', {METHOD => 'set', 'memo' =>
'big data'}
0 row(s) in 0.0930 seconds
```

```
hbase(main):025:0> alter_namespace 'bigdata', {METHOD => 'set', 'createtime'
=> '2016-01-30'}
0 row(s) in 0.0330 seconds

hbase(main):026:0>  describe_namespace 'bigdata'
DESCRIPTION
{NAME => 'bigdata', createtime => '2016-01-30', memo => 'big data'}
1 row(s) in 0.0100 seconds
```

要取消已设定的属性，规则为{METHOD=>'unset', NAME=>'key'}，指定取消的属性名，填充到 key：

```
hbase(main):029:0> alter_namespace 'bigdata', {METHOD => 'unset', NAME =>
'memo'}
0 row(s) in 0.0250 seconds

hbase(main):030:0>  describe_namespace 'bigdata'
DESCRIPTION
{NAME => 'bigdata', createtime => '2016-01-30'}
1 row(s) in 0.0120 seconds
```

(6) 删除命名空间。

drop_namespace 命令后跟要删除的命名空间，执行成功后，即可以删除指定的命名空间。这里需要注意的是，删除前，命名空间里面要没有表，如果命名空间有表，则先要手动将表删除后，再执行删除命名空间的操作。

这里我们删掉命名空间 big：

```
hbase(main):036:0> drop_namespace 'big'
0 row(s) in 0.0160 seconds

hbase(main):038:0> list_namespace
NAMESPACE
bigdata
default
hbase
3 row(s) in 0.0140 seconds
```

6.3.3 表管理

HBase 的表管理对应于前面提到的 DDL 命令，它定义了对表及其结构的操作，包括表的创建、结构修改、表的停用、表的删除、批量停用与删除表等。

表 6-5 给出了 DDL 命令清单。

表 6-5 DDL 命令清单

命令名	作用
alter	修改表，包括增加、删除列簇、修改压缩算法
alter_async	修改表的属性，如 MAX_FILESIZE、MEMSTORE_FLUSHSIZE、READONLY 和 DEFERRED_LOG_FLUSH 等
alter_status	修改表的状态，如修改表区域
create	创建表，指定表的列簇、表名、命名空间创建一张表
describe	描述表，查看指定表的结构

续表

命 令 名	作　用
disable	停用表，将指定表停止使用
disable_all	停用指定集合的所有表，支持以正则表达式获取表集合
drop	删除表，删除指定的表，前提是先将表停用
enable	启用表，将指定的表启用
drop_all	删除指定集合的所有表，支持以正则表达式获取表集合
enable_all	启用指定集合的所有表，支持以正则表达式获取表集合
exists	查看表是否存在
get_table	获取指定表的指针
is_disabled	判断表是否停用
is_enabled	判断表是否启用
list	列出指定表的信息
show_filters	查看过滤器

(1)　创建表。

使用 create 命令能够创建一张表，需要指定的参数包括表名称、列簇名称、命名空间，它的操作语法可以是如下几种。

①　create ‘表名称’, ‘列簇名称 1’, ‘列簇名称 2’, ‘列簇名称 N’。

②　create ‘命名空间:表名称’, ‘列簇名称 1’, ‘列簇名称 2’, ‘列簇名称 N’。

③　create ‘表名称’, {NAME => ‘列簇名称 1’, NAME => ‘列簇名称 2’, VERSIONS => 1, TTL => 2592000, BLOCKCACHE => true}，可以指定表的参数信息，如版本数量、TTL 等。

默认创建的表归属于 default 命名空间，这里用后两种方法创建两张表。一张是员工信息表 emp_tb，归属于 bigdata 命名空间，有三个列簇，一个是基本信息 basic，一个是收入信息 income，一个是工作信息 work；另一张是员工考勤表 emp_sign_tb，有一个列簇，考勤信息 sign。

```
hbase(main):012:0> create 'bigdata:emp_tb','basic','income','work'
0 row(s) in 1.6340 seconds

=> Hbase::Table - bigdata:emp_tb
hbase(main):013:0> create 'bigdata:emp_sign_tb',
 {NAME=>'sign',VERSIONS=>'3'}
0 row(s) in 0.2210 seconds

=> Hbase::Table - bigdata:emp_sign_tb
hbase(main):014:0> list_namespace_tables 'bigdata'
TABLE
emp_sign_tb
emp_tb
2 row(s) in 0.0190 seconds
```

通过 describe 命令，能够查看表的结构。查看上面创建成功的两张表的结构以及当前表的状态信息，如图 6-12 所示。

```
hbase(main):025:0> describe 'bigdata:emp_sign_tb'
DESCRIPTION                                                                                   ENABLED
 'bigdata:emp_sign_tb', {NAME => 'sign', BLOOMFILTER => 'ROW', VERSIONS => '3', IN_MEMORY => 'false', KEEP_DELETED_CEL true
 LS => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACH
 E => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
1 row(s) in 0.0290 seconds

hbase(main):026:0> describe 'bigdata:emp_tb'
DESCRIPTION                                                                                   ENABLED
 'bigdata:emp_tb', {NAME => 'basic', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS = true
 > 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE =>
  'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}, {NAME => 'income', BLOOMFILTER => 'ROW', VERSIONS => '1', I
 N_MEMORY => 'false', KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => '
 NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}, {NAME => 'work', B
 LOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NO
 NE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLIC
 ATION_SCOPE => '0'}
1 row(s) in 0.0360 seconds
```

图 6-12　查看表的结构信息

通过图 6-12 可以看出 HBase 对于表的元数据描述的格式，核心是基于列簇的描述，每个列簇有若干属性信息。以 emp_tb 表的 basic 列簇为示例，其列簇中所涉及的属性信息以及默认的值如下所示：

```
{NAME => 'basic',
BLOOMFILTER => 'ROW',
VERSIONS => '1',
IN_MEMORY => 'false',
KEEP_DELETED_CELLS => 'false',
DATA_BLOCK_ENCODING => 'NONE',
TTL => 'FOREVER',
COMPRESSION => 'NONE',
MIN_VERSIONS => '0',
BLOCKCACHE => 'true',
BLOCKSIZE => '65536',
REPLICATION_SCOPE => '0'}
```

表 6-6 给出了这些属性信息的作用，了解到这些作用后，可以在实际工作中根据需求去进行设置。

表 6-6　表列簇的参数信息

属 性 名	作　用
NAME	列簇名称
BLOOMFILTER	启用何种方式做布隆过滤，可以不启用，填充 NONE
VERSIONS	版本数量，存储份数，默认为 1
IN_MEMORY	数据保存在内存，用于提升性能，true 为保存，false 为不保存
KEEP_DELETED_CELLS	保留被删除单元格的数据，true 为保留，false 为不保留
DATA_BLOCK_ENCODING	数据块编码方式
TTL	数据生存时间，单位为秒，默认为永不过期
COMPRESSION	压缩算法，默认无压缩，为提高存储效率，可用 Snappy 压缩算法
MIN_VERSIONS	最少版本数量，默认为 0
BLOCKCACHE	是否启用存储块缓存
BLOCKSIZE	文件存储块的大小
REPLICATION_SCOPE	数据复制范围，默认为 0，为禁止；为 1 时，可向指定 HBase 集群进行数据同步复制

小提示

TTL 与 MIN_VERSIONS

① TTL 为 0，若 MIN_VERSIONS 为 0，则到期后，HBase 为自动删除本单元格数据，因此设置时要慎重。

② TTL 为 0，若 MIN_VERSIONS 为 1，则到期后，默认为保留时间最新的数据。

(2) 修改表。

通过 alter 等命令，可以修改表的结构，修改的范围，包括对表的属性信息、表列簇的新增、删除等。

它的语法如下所示。

① 取消某一属性：alter '命名空间:表名称', METHOD => 'table_att_unset', NAME => '属性名称'。

② 设定某一属性：alter '命名空间:表名称', METHOD => 'table_att', NAME => '属性名称'。

③ 修改表的某属性：alter '命名空间:表名称', 属性名。

④ 修改表列簇：alter '命名空间:表名称', NAME => '属性名 1 或列簇名称 1', METHOD =>'操作符', NAME => '属性名 2 或列簇名称 2', METHOD =>'操作符'。

这里针对 emp_tb 表中的 basic 列簇，将其版本数量调整为 3，TTL 值调整为 1000 秒，如下所示：

```
hbase(main):002:0> alter
'bigdata:emp_tb',NAME=>'basic',VERSIONS=>'3',TTL=>'2000'
Updating all regions with the new schema...
0/1 regions updated.

1/1 regions updated.
Done.
0 row(s) in 2.6370 seconds
```

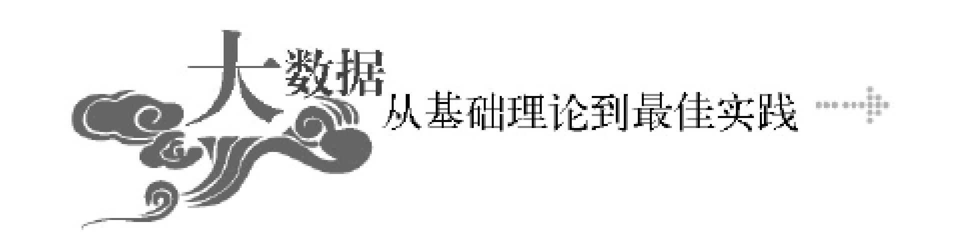
变更后的效果如图 6-13 所示。

```
hbase(main):008:0> describe 'bigdata:emp_tb'
DESCRIPTION                                                                                      ENABLED
 'bigdata:emp_tb', {NAME => 'basic', BLOOMFILTER => 'ROW', VERSIONS => '3', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'fa true
 lse', DATA_BLOCK_ENCODING => 'NONE', TTL => '2000 SECONDS (33 MINUTES 20 SECONDS)', COMPRESSION => 'NONE', MIN_VERSIONS =>
  '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}, {NAME => 'income', BLOOMFILTER => 'ROW', VERS
 IONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRES
 SION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}, {NAME => 'work
 ', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NON
 E', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_
 SCOPE => '0'}
1 row(s) in 0.0530 seconds
```

图 6-13　修改后的表结构

也可以新增一个列簇，这里针对 emp_sign_tb 表新增一个列簇 test，如下所示：

```
hbase(main):010:0> alter 'bigdata:emp_sign_tb',NAME=>'test'
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.1680 seconds
```



变更后的效果如图 6-14 所示。

```
hbase(main):011:0> describe 'bigdata:emp_sign_tb'
DESCRIPTION                                                                                          ENABLED
 'bigdata:emp_sign_tb', {NAME => 'sign', BLOOMFILTER => 'ROW', VERSIONS => '3', IN_MEMORY => 'false', KEEP_DELETED_CELLS => true
  'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true
 ', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}, {NAME => 'test', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => '
 false', KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSION
 S => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
1 row(s) in 0.0350 seconds
```

图 6-14　修改后的表结构

删除 emp_sign_tb 表新增的列簇 test，如下所示：

```
hbase(main):012:0> alter
'bigdata:emp_sign_tb',NAME=>'test',METHOD=>'delete'
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 1.1500 seconds
```

变更后的效果如图 6-15 所示。

```
hbase(main):013:0> describe 'bigdata:emp_sign_tb'
DESCRIPTION                                                                                          ENABLED
 'bigdata:emp_sign_tb', {NAME => 'sign', BLOOMFILTER => 'ROW', VERSIONS => '3', IN_MEMORY => 'false', KEEP_DELETED_CELLS => true
  'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true
 ', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
1 row(s) in 0.0380 seconds
```

图 6-15　修改后的表结构

可以修改表的属性，如修改表变成只读，修改表的范围等。例如，将 emp_sign_tb 表变成只读，设定一个列簇最大存储空间为 128MB，相关的操作如下所示：

```
hbase(main):039:0> alter_async 'bigdata:emp_sign_tb', READONLY
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.1140 seconds
hbase(main):041:0> alter 'bigdata:emp_sign_tb', METHOD => 'table_att',
MAX_FILESIZE => '134217728'
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.1390 seconds
```

变更后的效果如图 6-16 所示。

```
hbase(main):042:0> describe 'bigdata:emp_sign_tb'
DESCRIPTION                                                                                          ENABLED
 'bigdata:emp_sign_tb', {TABLE_ATTRIBUTES => {MAX_FILESIZE => '134217728'}, {NAME => 'READONLY', BLOOMFILTER => 'ROW', VERS true
 IONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRES
 SION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}, {NAME => 'sign
 ', BLOOMFILTER => 'ROW', VERSIONS => '3', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NON
 E', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_
 SCOPE => '0'}
1 row(s) in 0.0460 seconds
```

图 6-16　修改后的表结构

(3)　表状态控制。

通过 enable、disable 命令，可以控制表的状态，如一张表的启用或停用；enable_all、

disable_all 命令可针对一组表进行控制，这两个命令支持正则表达式。

例如，将表 emp_sign_tb 停用，然后再启用，操作如下：

```
hbase(main):006:0> disable 'bigdata:emp_sign_tb'
0 row(s) in 1.6930 seconds
hbase(main):008:0> is_disabled 'bigdata:emp_sign_tb'
true
0 row(s) in 0.0310 seconds
hbase(main):009:0> enable 'bigdata:emp_sign_tb'
0 row(s) in 0.2550 seconds
hbase(main):010:0> is_disabled 'bigdata:emp_sign_tb'
false
0 row(s) in 0.0210 seconds
```

如果针对一组表的操作，采用正则表达式，如将前缀为 emp 的所有表停用，然后再启用，操作如下(这里注意正则表达式的写法是 emp.*)：

```
hbase(main):035:0> disable_all 'bigdata:emp.*'
bigdata:emp_sign_tb
bigdata:emp_tb

Disable the above 2 tables (y/n)? y
2 tables successfully disabled

hbase(main):043:0> enable_all 'bigdata:emp.*'
bigdata:emp_sign_tb
bigdata:emp_tb

Enable the above 2 tables (y/n)? y
2 tables successfully enabled
```

小提示

表的修改。

① 对表进行 alter 修改时，通常建议先停用该表，完成修改后再启用。

② 表在修改前，对重要数据做好备份是良好的习惯。

(4) 表清单。

list 可列出 HBase 的所有用户数据表清单，默认执行时，会列出全部的用户数据表，由于 list 命令支持正则表达式，因此，也可以通过正则语法获取满足条件的表清单。

例如，使用 list 列出全部表，使用 list 的正则模式，列出命名空间 bigdata 中所有的表：

```
hbase(main):121:0> list
TABLE
bigdata:emp_sign_tb
bigdata:emp_tb
testhbase
3 row(s) in 0.0090 seconds

=> ["bigdata:emp_sign_tb", "bigdata:emp_tb", "testhbase"]
hbase(main):122:0> list 'bigdata:.*'
TABLE
bigdata:emp_sign_tb
bigdata:emp_tb
2 row(s) in 0.0090 seconds
```

```
=> ["bigdata:emp_sign_tb", "bigdata:emp_tb"]
```

(5) 删除表。

命令 drop、drop_all 可删除表，表被删除的前提是已被停用。还有一条命令 get_table 用于获取表的操作句柄，类似于表指针，能够简化实际命令操作。

例如，新建一张表 test，然后借助于 get_table 指向 tt，由 tt 指针完成表的停用、删除操作，如下所示：

```
hbase(main):044:0> create 'bigdata:test','f1'
0 row(s) in 0.4530 seconds

=> Hbase::Table - bigdata:test
hbase(main):045:0> tt = get_table 'bigdata:test'
0 row(s) in 0.0040 seconds

=> Hbase::Table - bigdata:test

hbase(main):049:0> tt.disable
0 row(s) in 1.3200 seconds

hbase(main):050:0> tt.drop
0 row(s) in 0.2290 seconds

hbase(main):052:0> exists 'bigdata:test'
Table bigdata:test does not exist
0 row(s) in 0.0240 seconds
```

6.4　HBase 的数据管理

数据管理对应于 HBase 的 DML 命令，它定义了对数据的添加、修改、删除、查询、统计等操作。表 6-7 给出了 DML 命令清单。

表 6-7　DML 命令清单

命 令 名	作　用
append	对指定的单元格的值后面追加数据
count	统计指定表、行、列的记录数
delete	删除指定的数据单元
deleteall	删除指定的行
get	获得指定的数据单元或者行的内容
get_counter	获取指定单元格的计数器的值
incr	对指定单元格进行自增
put	提交单元格数据
scan	全表扫描查找数据
truncate	先删除指定的表再重建
truncate_preserve	保留原表区域的前提下删除指定的表再重建

6.4.1　数据的添加

put 命令可用于添加一个单元格的数据，它的使用原理是通过表名、行键值、列名定位到一个单元格，为这个单元格填充数据。

它的语法如下所示。

(1)　put ‘命名空间:表名称’, ‘行键值’, ‘列簇:列名’, ‘列值’。

(2)　put ‘命名空间:表名称’, ‘行键值’, ‘列簇:列名’, ‘列值’, 时间戳。

(3)　put ‘命名空间:表名称’, ‘行键值’, ‘列簇:列名’, ‘列值’, {ATTRIBUTES=>{‘自定义键值’=>‘自定义值’}}。

(4)　t.put ‘行键值’, ‘列簇:列名’, ‘列值’。(t 代表获取表的指针)

例如，向 emp_sign_tb 表添加如表 6-8 所示的数据。

表 6-8　员工出勤表

emp_sign_tb 表					
行 键 值 (Row Key) (员工姓名)	时 间 戳 (TimeStamp)	sign 列簇(column family)			
		F201601	F201602	F201603	…
standard	20160201	25			
	20160301		20		
Tom	20160201	20			
	20160301		19		
Jim	20160201	21			
	20160301		20		

该表中的行键值为姓名字段；每月实际出勤次数字段，以 Fyyyymm 的格式进行动态增加，yyyy 代表年份，mm 代表月份。其中第一行的 standard 代表当月的法定出勤工作日。

根据上述信息，操作命令如下。

```
hbase(main):027:0> t = get_table 'bigdata:emp_sign_tb'
0 row(s) in 0.0330 seconds

=> Hbase::Table - bigdata:emp_sign_tb
hbase(main):029:0> t.put 'standard','sign:F201601','25',20160201
0 row(s) in 0.7460 seconds
hbase(main):030:0> t.put 'standard','sign:F201602','20',20160301
0 row(s) in 0.0130 seconds
hbase(main):031:0> t.put 'Tom','sign:F201601','20',20160201
0 row(s) in 0.0070 seconds
hbase(main):032:0> t.put 'Tom','sign:F201602','19',20160301
0 row(s) in 0.0060 seconds
hbase(main):033:0> t.put 'Jim','sign:F201601','21',20160201
0 row(s) in 0.0130 seconds
hbase(main):034:0> t.put 'Jim','sign:F201602','20',20160301
0 row(s) in 0.0040 seconds
通过 scan 命令，查看 emp_sign_tb 表的所有记录。
hbase(main):044:0> t.scan
```

```
ROW                          COLUMN+CELL
 Jim                         column=sign:F201601, timestamp=20160201,
value=21
 Jim                         column=sign:F201602, timestamp=20160301,
value=20
 Tom                         column=sign:F201601, timestamp=20160201,
value=20
 Tom                         column=sign:F201602, timestamp=20160301,
value=19
 standard                     column=sign:F201601, timestamp=20160201,
value=25
 standard                     column=sign:F201602, timestamp=20160301,
value=20
3 row(s) in 0.0580 seconds
```

小提示

关于本例中的 put 命令。

① 本例中的列是动态增加的，且使用时不需要提前声明。

② 行键值的选取要极为慎重，防止出现重复等情况。

③ 时间戳不需要用引号括起来，如果不填充，系统会自动添加。

6.4.2 数据的追加

append 命令用于向指定的单元格后面追加数据，如果上单元格不存在，则直接添加；如果存在，则将数据追加到原值后面。

它的语法如下。

(1) append '命名空间:表名称', '行键值', '列簇:列名', '列值'。

(2) append '命名空间:表名称', '行键值', '列簇:列名', '列值', {ATTRIBUTES=>{'自定义键值'=>'自定义值'}}。

(3) t. append '行键值', '列簇:列名', '列值' (t 代表获取表的指针)。

为了方便理解，这里安排一个这样的示例，向 emp_sign_tb 表加入一条新的用户记录 Jack，他的 201601 考勤是 21 天，201602 的考勤是 18 天，由于添加失误，他的 201601 写成了 2 天，201602 写成了 1 天。这里通过 append 命令进行纠正，分别再添加 0、8。

操作如下所示：

```
hbase(main):050:0> t.put 'Jack','sign:F201601','2',20160201
0 row(s) in 0.0070 seconds
hbase(main):051:0> t.put 'Jack','sign:F201602','1',20160301
0 row(s) in 0.0070 seconds
hbase(main):052:0> t.scan
ROW                          COLUMN+CELL
 Jack                        column=sign:F201601, timestamp=20160201,
value=2
 Jack                        column=sign:F201602, timestamp=20160301,
value=1
...
hbase(main):053:0> t.append 'Jack','sign:F201601','0'
0 row(s) in 0.0080 seconds
hbase(main):054:0> t.append 'Jack','sign:F201602','8'
0 row(s) in 0.0040 seconds
```

```
hbase(main):055:0> t.scan
ROW                          COLUMN+CELL
 Jack                         column=sign:F201601,
timestamp=1455696906196, value=20
 Jack                         column=sign:F201602,
timestamp=1455696913859, value=18
 ...
```

小提示

关于 append。

① append 针对数据的修改提供了一种追加的方法。

② 数据追加后，要注意时间戳默认会修改。

6.4.3　数据的获取

get 命令通过指定的表、行、列簇或列获取某一行或某一列簇、某一单元格的数据，get 命令对应于 put 命令，均属于针对单行的数据操作。

它的基本语法如下所示：

- get ‘命名空间:表名’, ‘行键值’, ‘列簇’。
- get ‘命名空间:表名’, ‘行键值’, ‘列簇:列’。
- get ‘命名空间:表名’, ‘行键值’, {COLUMN => ‘列簇:列’, TIMERANGE => [ts1, ts2], VERSIONS => 2}。
- get ‘命名空间:表名’, ‘行键值’, {FILTER => “ValueFilter(=, ‘binary:abc’)”}。
- get ‘命名空间:表名’, ‘行键值’, {COLUMN => [‘列簇 1:列 1’, ‘列簇 2:列 1’], ATTRIBUTES => {‘自定义键值’=>‘自定义数据’}}。
- t.get ‘行键值’, ‘列簇’。

从语法中可以看出，除了表、行、列簇、列之外，还可以通过时间戳、版本号、过滤器、自定义标签等作为检索条件，获取满足要求的数据。

(1)　获取行数据。

通过行键值，可以获取指定行的全部列簇数据，这里获取 emp_sign_tb 表中的 Jack 的全部考勤记录：

```
hbase(main):001:0> t = get_table 'bigdata:emp_sign_tb'
0 row(s) in 0.5270 seconds

=> Hbase::Table - bigdata:emp_sign_tb
hbase(main):020:0> t.get 'Jack'
COLUMN                        CELL
 sign:F201601                  timestamp=1455696906196, value=20
 sign:F201602                  timestamp=1455696913859, value=18
2 row(s) in 0.0230 seconds
```

(2)　获取列簇数据。

通过参数直接指定行键值，会列出该行所有的记录，在有的时候，一张表可能包括多个列簇，不需要列出全部的列簇，这里也可以通过加入列簇参数进行控制，显示关注的列簇，如下所示，只关注表 sign 列簇中的所有数据：

```
hbase(main):021:0> t.get 'Jack','sign'
COLUMN                          CELL
 sign:F201601                    timestamp=1455696906196, value=20
 sign:F201602                    timestamp=1455696913859, value=18
2 row(s) in 0.0180 seconds
```

(3) 获取列数据。

同理，也可以只关注于列簇中某一列，或者某几列的数据。例如，关注某一列的数据操作，如下所示：

```
hbase(main):029:0> t.get 'Jack','sign:F201601'
COLUMN                          CELL
 sign:F201601                    timestamp=1455696906196, value=20
1 row(s) in 0.0120 seconds
```

(4) 获取列集合数据。

关注某几列的信息，通过[]符号将关注的列清单括起来。例如，可能需要关注不同列簇的几个列，可以将其组成一个集合进行获取，如下所示：

```
hbase(main):030:0> t.get
'Jack',['sign:F201601','sign:F201602','sign:F201603']
COLUMN                          CELL
 sign:F201601                    timestamp=1455696906196, value=20
 sign:F201602                    timestamp=1455696913859, value=18
2 row(s) in 0.0290 seconds
```

(5) 获取多版本数据。

在获取数据时，也可以通过加入版本这个过滤条件，获取最近几次的更新情况。例如，针对 Jack 这一行，做过两次数据提交，可以查看历次提交的信息，进行比对：

```
hbase(main):058:0> t.get 'Jack',{COLUMN=>['sign:F201601'],VERSIONS=>2}
COLUMN                          CELL
 sign:F201601                    timestamp=1455696906196, value=20
 sign:F201601                    timestamp=20160201, value=2
2 row(s) in 0.0120 seconds
```

小提示

关于多版本的意义。

① 利于数据的安全性审计。保留某单元格的多版本数据，可以事后审计，发现有无非法的修改。

② 利于数据的灾备恢复与回滚。可以列出最近几次的修改，若发现最新数据更新不正确，则可以回滚到最近一次或最近几次的修改。

(6) 通过时间戳获取数据。

通过单个时间戳，可以获取该行在这一时间点上修改的所有列，这里获取时间戳为 201601 的 Jack 数据：

```
hbase(main):067:0> t.get 'Jack',{TIMESTAMP=>20160201}
COLUMN                          CELL
 sign:F201601                    timestamp=20160201, value=2
1 row(s) in 0.0130 seconds
```

通过时间区间，可以获取一定时间范围内的该行列数据，需要使用的关键字是 TIMERANGE。这里获取 20160201、20160304 之间的数据，如下所示：

```
hbase(main):066:0> t.get 'Jack',{TIMERANGE=>[20160201,2016020304]}
COLUMN                          CELL
 sign:F201601                    timestamp=20160201, value=2
 sign:F201602                    timestamp=20160301, value=1
2 row(s) in 0.0120 seconds
```

(7) 通过过滤器获取数据。

HBase 中提供了过滤器功能，过滤器属于对数据结果集合的符合性筛选，这对于海量信息查询与分析非常重要，引入过滤器，可以更丰富地控制数据结果的抽取。

命令 show_filters 可以查看 HBase 当前支持的过滤器清单，如下所示：

```
hbase(main):084:0> show_filters
...
DependentColumnFilter
KeyOnlyFilter
ColumnCountGetFilter
SingleColumnValueFilter
PrefixFilter
SingleColumnValueExcludeFilter
FirstKeyOnlyFilter
ColumnRangeFilter
TimestampsFilter
FamilyFilter
QualifierFilter
ColumnPrefixFilter
RowFilter
MultipleColumnPrefixFilter
InclusiveStopFilter
PageFilter
ValueFilter
ColumnPaginationFilter
```

通过 HBase 自带的过滤器，可以实现<行、列簇、列、值>多个维度的数据筛选操作。结合 show_filters 的输出，表 6-9 给出了这些过滤器的作用。

表 6-9　过滤器的作用

名　称	作　用
DependentColumnFilter	以某一列为参照，去过滤其他列
KeyOnlyFilter	返回表中每行的行键值
ColumnCountGetFilter	返回表中每行最多返回多少列，当在一行中返回的列数超过设定的值时，则结束扫描
SingleColumnValueFilter	基于某一列的值来判断是否返回当前行的数据
PrefixFilter	基于行键值与特定前缀的匹配度，来确定该行是否返回数据
SingleColumnValueExcludeFilter	返回不符合过滤列条件的行数据
FirstKeyOnlyFilter	返回结果集中的第一列数据
ColumnRangeFilter	返回满足列区间条件的行数据
RowFilter	返回满足条件的所有行数据

续表

名　称	作　用
MultipleColumnPrefixFilter	返回满足多个列前缀的行数据
InclusiveStopFilter	设置开始与结束行键值，返回这个区间内的行数据
PageFilter	设置行分页提升查询性能，每次返回指定数量的记录数据
ValueFilter	以具体的值来筛选单元格
ColumnPaginationFilter	设置列分页，提升查询性能，每次返回指定数量的列数据

下面的示例中，采用 ValueFilter 过滤器，通过该过滤器来列出 Jack 完成 20 天考勤的月份清单，如下所示：

```
hbase(main):086:0> t.get 'Jack',{FILTER => "ValueFilter(=, 'binary:20')"}
COLUMN                          CELL
 sign:F201601                    timestamp=1455696906196, value=20
1 row(s) in 0.0120 seconds
```

6.4.4　数据统计

count 命令用于记录行数的统计，使用方法比较简单，指定表名，可以统计该表拥有多少行，如下所示：

```
hbase(main):144:0> t.count
5 row(s) in 0.0050 seconds
=> 5
```

对于大部分 HBase 业务数据表而言，其记录是海量的，因此，在执行 count 命令对全表统计时会非常耗时、耗资源，在 count 命令中也提供了两个优化选项，一个是 interval，即设置统计多少行后显示一次对应的 rowkey，这个参数默认值 1000；另外一个是 cache，即每次去取的缓存行数，默认为 10 条。

示例中，分别调整[interval=>1, cache=>1]、[interval=>2, cache=>5]查看各自统计的结果，如下所示：

```
hbase(main):145:0> t.count INTERVAL=>1,CACHE=>1
Current count: 1, row: Jack
Current count: 2, row: Jim
Current count: 3, row: Tom
Current count: 4, row: standard
Current count: 5, row: tt
5 row(s) in 0.0090 seconds
=> 5
hbase(main):150:0> t.count INTERVAL=>2,CACHE=>5
Current count: 2, row: Jim
Current count: 4, row: standard
5 row(s) in 0.0080 seconds
=> 5
```

小提示

关于统计性能。

① 使用 count 命令，如果表中各行空间占用不多，可以提高 cache 大小，提高性能；

另外，适度调整 interval 值，可以获取性能提升。

② 在创建表时指定 Coprocessor 参数，可以驱动各 RegionServer 分头统计目标表各 Region 中的行数，能够比较快地获取结果。

③ 对于不频繁的统计任务，可以编写 MapReduce 任务去执行。

6.4.5　表的扫描

scan 用于全表数据扫描，可以返回一张表的所有数据，也可以根据多维过滤条件，如 TIMERANGE、FILTER、LIMIT、STARTROW、STOPROW、TIMESTAMP、MAXLENGTH、COLUMNS、CACHE 等，返回指定的结果集。

它的基本语法如下所示：

- scan ‘命名空间:表名称’。
- scan ‘命名空间:表名称’, {COLUMNS => ‘列簇:列’}。
- scan ‘命名空间:表名称’, {COLUMNS => [‘列簇:列’, ‘列簇:列’], LIMIT => 10, STARTROW => ‘xyz’}。
- scan ‘命名空间:表名称’, {FILTER => ColumnPaginationFilter.new(1, 0)}。
- scan ‘命名空间:表名称’, {COLUMNS => ‘列簇:列’, TIMERANGE => [1303668814, 1303669904]}。

(1) 获取行数据。

通过表名称，返回全部记录，这里获取 emp_sign_tb 表中的所有用户的考勤记录，如下所示：

```
hbase(main):019:0> scan 'bigdata:emp_sign_tb'
ROW                               COLUMN+CELL
 Jack                 column=sign:F201601, timestamp=1455696906196, value=20
 Jack                 column=sign:F201602, timestamp=1455696913859, value=18
 Jim                  column=sign:F201601, timestamp=20160201, value=21
 Jim                  column=sign:F201602, timestamp=20160301, value=20
 Tom                  column=sign:F201601, timestamp=20160201, value=20
 Tom                  column=sign:F201602, timestamp=20160301, value=19
 standard              column=sign:F201601, timestamp=20160201, value=25
 standard              column=sign:F201602, timestamp=20160301, value=20
4 row(s) in 0.0260 seconds
```

(2) 获取列数据。

获取表中指定列的数据，这里获取 sign 列簇中 201602 的考勤记录，如下所示：

```
hbase(main):021:0> scan 'bigdata:emp_sign_tb',{COLUMN=>'sign:F201602'}
ROW                    COLUMN+CELL
 Jack                  column=sign:F201602, timestamp=1455696913859, value=18
 Jim                   column=sign:F201602, timestamp=20160301, value=20
 Tom                   column=sign:F201602, timestamp=20160301, value=19
 standard              column=sign:F201602, timestamp=20160301, value=20
4 row(s) in 0.0250 seconds
```

(3) 限制输出行数。

通过 limit n 参数，可以设置输出的行数，只限于前 n 行记录返回，这里返回表中 201601、201602 两个月份的前 2 行记录，如下所示：

```
hbase(main):023:0> scan
'bigdata:emp_sign_tb',{COLUMN=>['sign:F201601','sign:F201602'],LIMIT=>2}
```

```
ROW                    COLUMN+CELL
 Jack                  column=sign:F201601, timestamp=1455696906196, value=20
 Jack                  column=sign:F201602, timestamp=1455696913859, value=18
 Jim                   column=sign:F201601, timestamp=20160201, value=21
 Jim                   column=sign:F201602, timestamp=20160301, value=20
2 row(s) in 0.0390 seconds
```

(4) 获取多版本数据。

通过 VERSIONS 参数，可以获取多版本数据，这里返回前 2 行，每行 3 个最近版本的数据信息：

```
hbase(main):026:0> scan
'bigdata:emp_sign_tb',{COLUMN=>['sign:F201601','sign:F201602'],
LIMIT=>2,VERSIONS=>3}
ROW                    COLUMN+CELL
 Jack                  column=sign:F201601, timestamp=1455696906196, value=20
 Jack                  column=sign:F201601, timestamp=20160201, value=2
 Jack                  column=sign:F201602, timestamp=1455696913859, value=18
 Jack                  column=sign:F201602, timestamp=20160301, value=1
 Jim                   column=sign:F201601, timestamp=20160201, value=21
 Jim                   column=sign:F201602, timestamp=20160301, value=20
2 row(s) in 0.0240 seconds

hbase(main):027:0>
```

(5) 获取逆向排序数据。

通过 REVERSED 参数，可以控制结果集正向还是逆向输出，默认输出方式基于字母表顺序正序输出，如果将 REVERSED 置为 true，则输出变成逆向。这里输出全表，读者可以与前面的 scan 'bigdata:emp_sign_tb'命令结果进行对比：

```
hbase(main):030:0> scan 'bigdata:emp_sign_tb',{REVERSED=>TRUE}
ROW                    COLUMN+CELL
 standard               column=sign:F201601, timestamp=20160201, value=25
 standard               column=sign:F201602, timestamp=20160301, value=20
 Tom                   column=sign:F201601, timestamp=20160201, value=20
 Tom                   column=sign:F201602, timestamp=20160301, value=19
 Jim                   column=sign:F201601, timestamp=20160201, value=21
 Jim                   column=sign:F201602, timestamp=20160301, value=20
 Jack                  column=sign:F201601, timestamp=1455696906196, value=20
 Jack                  column=sign:F201602, timestamp=1455696913859, value=18
5 row(s) in 0.0260 seconds
```

(6) 通过时间戳获取数据。

通过设定时间区间，用于获取满足条件的所有记录。这里获取 20160301、20160304 之间的数据，如下所示：

```
hbase(main):033:0> scan
'bigdata:emp_sign_tb',{REVERSED=>FALSE,TIMERANGE=>[20160
301,2016020304]}
ROW                    COLUMN+CELL
 Jack                  column=sign:F201602, timestamp=20160301, value=1
 Jim                   column=sign:F201602, timestamp=20160301, value=20
 Tom                   column=sign:F201602, timestamp=20160301, value=19
 standard               column=sign:F201602, timestamp=20160301, value=20
4 row(s) in 0.0120 seconds
```

(7) 通过过滤器获取数据。

采用 FILTER，可以非常方便地查询结果集。这里以 PrefixFilter 过滤器为例，返回首字母为 J 的所有员工考勤信息，如下所示：

```
hbase(main):038:0> scan 'bigdata:emp_sign_tb',{FILTER => "(PrefixFilter
('J')"}
```

```
ROW                   COLUMN+CELL
 Jack                 column=sign:F201601, timestamp=1455696906196, value=20
 Jack                 column=sign:F201602, timestamp=1455696913859, value=18
 Jim                  column=sign:F201601, timestamp=20160201, value=21
 Jim                  column=sign:F201602, timestamp=20160301, value=20
2 row(s) in 0.0190 seconds
```

6.4.6　数据的删除

delete 命令用于删除某一单元格数据，删除时，需要指定表名、行键值、列名，以及时间戳，即可以精确控制删除本列中某一版本的数据。

例如，表 emp_sign_tb 的 Jack 行中 sign:F201601 列的 20160201 时间戳单元格数据有问题要删除，这里执行删除操作，如下所示：

```
hbase(main):043:0> scan 'bigdata:emp_sign_tb',{VERSIONS=>3}
ROW                            COLUMN+CELL
 Jack       column=sign:F201601, timestamp=1455696906196, value=20
 Jack       column=sign:F201601, timestamp=20160201, value=2
 Jack       column=sign:F201602, timestamp=1455696913859, value=18
 Jack       column=sign:F201602, timestamp=20160301, value=1
 ...
hbase(main):044:0> delete
'bigdata:emp_sign_tb','Jack','sign:F201601',20160201
0 row(s) in 0.0840 seconds

hbase(main):045:0> scan 'bigdata:emp_sign_tb',{VERSIONS=>3}
ROW                   COLUMN+CELL
 Jack                 column=sign:F201601, timestamp=1455696906196, value=20
 Jack                 column=sign:F201602, timestamp=1455696913859, value=18
 Jack                 column=sign:F201602, timestamp=20160301, value=1
...
```

deleteall 命令可以删除整行，通过指定表名、行键值，即可完成删除操作，这里假设 Jim 员工离职，需要删除其所有考勤记录，操作如下：

```
hbase(main):046:0> deleteall 'bigdata:emp_sign_tb','Jim'
0 row(s) in 0.0100 seconds
hbase(main):047:0> scan 'bigdata:emp_sign_tb'
ROW                   COLUMN+CELL
 Jack                 column=sign:F201601, timestamp=1455696906196, value=20
 Jack                 column=sign:F201602, timestamp=1455696913859, value=18
 Tom                  column=sign:F201601, timestamp=20160201, value=20
 Tom                  column=sign:F201602, timestamp=20160301, value=19
 standard              column=sign:F201601, timestamp=20160201, value=25
 standard              column=sign:F201602, timestamp=20160301, value=20
3 row(s) in 0.0170 seconds
```

6.4.7　表的重建

(1)　truncate 命令用于表的重建，本命令重建的过程等价于：表的停用、表的删除、表的创建，这里以 emp_sign_tb 为例，重建这张表，操作如下所示：

```
hbase(main):048:0> truncate 'bigdata:emp_sign_tb'
Truncating 'bigdata:emp_sign_tb' table (it may take a while):
 - Disabling table...
 - Dropping table...
 - Creating table...
0 row(s) in 1.9760 seconds
```

```
hbase(main):049:0> scan 'bigdata:emp_sign_tb'
ROW                   COLUMN+CELL
0 row(s) in 0.0200 seconds
```

(2) 保留数据区域的重建。

truncate_preserve 也用于表的重建，但与 truncate 不同的是，执行本命令重建表时，会保留表原有的 Region。

以 emp_sign_tb 为例，在上一步重建后，当前 HBase 所拥有的总 Region 数目为 7，当前的 emp_sign_tb 的 Region 编号为 968593698f9ee4ec88af0daab8924b53，如图 6-17 所示。

Master iZ23d4by1laZ

Region Servers

Base Stats　Memory　Requests　Storefiles　Compactions

ServerName	Start time	Requests Per Second	Num. Regions
iZ23d4by1laZ,60020,1456904989052	Wed Mar 02 15:49:49 CST 2016	0	7
Total:1		0	7

Table Attributes

Attribute Name	Value	Description
Enabled	true	Is the table enabled
Compaction	NONE	Is the table compacting

Table Regions

Name	Region Server	Start Key	End Key	Requests
bigdata:emp_sign_tb,,1456922754094.968593698f9ee4ec88af0daab8924b53.	iZ23d4by1laZ:60020			0

Regions by Region Server

Region Server	Region Count
iZ23d4by1laZ:60020	1

图 6-17　重建前的状态

执行完表的数据添加，通过重建后，相关命令的运行效果如下所示：

```
hbase(main):017:0>  t = get_table 'bigdata:emp_sign_tb'
0 row(s) in 0.0060 seconds

=> Hbase::Table - bigdata:emp_sign_tb
hbase(main):018:0>  t.put 'xxx','sign:F201601','20',20160201
0 row(s) in 0.0260 seconds
hbase(main):019:0> t.scan
ROW                   COLUMN+CELL
 xxx                  column=sign:F201601, timestamp=20160201, value=20
1 row(s) in 0.0090 seconds
hbase(main):020:0> truncate_preserve 'bigdata:emp_sign_tb'
Truncating 'bigdata:emp_sign_tb' table (it may take a while):
 - Disabling table...
```

```
 - Dropping table...
 - Creating table with region boundaries...
0 row(s) in 1.6970 seconds
hbase(main):021:0> t.scan
ROW                                     COLUMN+CELL
0 row(s) in 0.0380 seconds
hbase(main):022:0>
```

表重建后，可以发现，Region 的数目没有变，而此时新的 emp_sign_tb 表的 Region 编号已变成 b67a4dea80dd7c3d2495fbb0c798bfb1，如图 6-18 所示。

Master iZ23d4by1laZ

Region Servers

Base Stats　Memory　Requests　Storefiles　Compactions

ServerName	Start time	Requests Per Second	Num. Regions
iZ23d4by1laZ,60020,1456904989052	Wed Mar 02 15:49:49 CST 2016	0	7
Total:1		0	7

Table Attributes

Attribute Name	Value	Description
Enabled	true	Is the table enabled
Compaction	NONE	Is the table compacting

Table Regions

Name	Region Server	Start Key	End Key	Requests
bigdata:emp_sign_tb,,1456922965530.b67a4dea80dd7c3d2495fbb0c798bfb1.	iZ23d4by1laZ:60020			0

Regions by Region Server

Region Server	Region Count
iZ23d4by1laZ:60020	1

图 6-18　重建后的状态

6.5　HBase 的集群管理

在生产环境中使用的多是 HBase 分布式集群，掌握如何自动化部署 HBase 集群以及如何管理与调度好集群是在实际项目中应用 HBase 的必要条件。

6.5.1　集群部署

HBase 集群需要部署到 Hadoop 之上，它的部署可以参考前面的 Hadoop 多节点部署，部署逻辑如下。

(1)　主机的配置：配置好各节点的操作系统、时间服务、网络、主机名、Java 环境、

SSH 环境。

(2) Hadoop 的配置：配置好各节点的 Hadoop 集群，并验证成功。

(3) Master 节点的配置：HBase 的 Master 节点的 HBase 安装，配置好 hbase-env.xml、hbase-site.xml、regionservers 等文件。

(4) RegionServer 节点的配置：将 Master 节点的 HBase 同步复制到各 RegionServer 节点，并针对性地调整配置。

(5) 启动与验证：在 Master 节点启动 HBase，验证安装配置是否成功。

这里以三节点为例，简要介绍 HBase 集群的部署。

1. 环境介绍

三节点的基本信息如表 6-10 所示。

表 6-10 各节点的信息

端 口 号	主 机 名	IP	作 用
节点 1	masterH	172.16.2.211	Master 节点/ RegionServer 节点
节点 2	slaveA	172.16.2.212	RegionServer 节点
节点 3	slaveB	172.16.2.213	RegionServer 节点

各节点首先需要配置时间服务器，确保时钟一致性，在这里，由 masterH 充当授时服务器，其他两个节点作为从节点进行时间同步。除时钟服务器外，还需要统一配置本地 DNS 映射，各节点的 DNS 信息如下所示：

```
[root@masterH conf]# vi  /etc/hosts
...
172.16.2.211        masterH
172.16.2.212        slaveA
172.16.2.213        slaveA
...
```

2. 修改 Master 配置文件

(1) hbase-env.sh 文件，设定环境变量，如下所示：

```
[root@masterH conf]# vi hbase-env.sh
...
export JAVA_HOME=/usr/java/jdk1.8.0_05/
export HBASE_MANAGES_ZK=true
...
```

设定 HBASE_MANAGES_ZK 的值为 true，确保 ZooKeeper 与 HBase 共启动，在本次集群部署时，每个节点都需要部署 ZooKeeper。

(2) hbase-site.xml 文件，具体配置信息如下：

```
[root@ masterH conf]# vi hbase-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
   <property>
      <name>hbase.master</name>
      <value> masterH:60000</value>
```

```
    </property>
    <property>
        <name>hbase.master.maxclockskew</name>
        <value>180000</value>
    </property>
    <property>
        <name>hbase.rootdir</name>
        <value> hdfs://masterH:9000/hbase </value>
    </property>
    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>
    <property>
        <name>hbase.Zookeeper.quorum</name>
        <value> masterH,slaveA,slaveB </value>
    </property>
    <property>
        <name>hbase.zookeeper.property.dataDir</name>
        <value>/home/hbase/zookeeper</value>
    </property>
</configuration>
```

重点配置 hbase.cluster.distributed、hbase.zookeeper.property.dataDir、hbase.rootdir、hbase.Zookeeper.quorum 四个参数，其中，hbase.cluster.distributed 设置为 true，用于支持分布式部署；hbase.zookeeper.property.dataDir 配置 ZooKeeper 集群的 data 目录，这里放置于 /home/hbase/zookeeper 目录，使用前，先确保目录创建完毕；hbase.rootdir 配置 HDFS 的集群地址，这里设定为 hdfs://masterH:9000/hbase，需要注意，在 Hadoop 文件系统里创建好 hbase 目录，并开放相关的权限；hbase.Zookeeper.quorum 配置 ZooKeeper，这里需要在三个节点均启动 ZooKeeper。

(3) regionservers 文件，用于配置哪些节点为 RegionServer，这里配置如下：

```
[root@masterH conf]# vi regionservers
masterH
slaveA
slaveB
```

3. 配置信息同步

通过 scp 命令，将 Master 节点的 HBase 目录同步至其他节点，如下所示：

```
[root@masterH home]# scp -r hbase  root@slaveA:/home/
[root@masterH home]# scp -r hbase  root@slaveB:/home/
```

4. 启动与验证

先启动 Hadoop 集群，验证无误后，启动 HBase，启动命令如下所示：

```
[root@masterH sbin]# ./start-all.sh
...
Starting namenodes on [masterH]
masterH: starting namenode, logging to
/home/hadoop/logs/hadoop-root-namenode-localhost.localdomain.out
masterH: starting datanode, logging to
/home/hadoop/logs/hadoop-root-datanode-localhost.localdomain.out
slaveB: starting datanode, logging to
/home/hadoop/logs/hadoop-root-datanode-localhost.localdomain.out
```

```
slaveA: starting datanode, logging to
/home/hadoop/logs/hadoop-root-datanode-localhost.localdomain.out
...
starting yarn daemons
starting resourcemanager, logging to
/home/hadoop/logs/yarn-root-resourcemanager-localhost.localdomain.out
slaveB:starting nodemanager,
logging to
/home/hadoop/logs/yarn-root-nodemanager-localhost.localdomain.out
masterH:starting nodemanager,
logging to
/home/hadoop/logs/yarn-root-nodemanager-localhost.localdomain.out
slaveA: starting nodemanager,
logging to
/home/hadoop/logs/yarn-root-nodemanager-localhost.localdomain.out
[root@ masterH bin]# ./start-hbase.sh
slaveA: starting zookeeper, logging to
/home/hbase/bin/../logs/hbase-root-zookeeper-localhost.localdomain.out
slaveB: starting zookeeper, logging to
/home/hbase/bin/../logs/hbase-root-zookeeper-localhost.localdomain.out
masterH: starting zookeeper,
logging to
/home/hbase/bin/../logs/hbase-root-zookeeper-localhost.localdomain.outst
arting master, logging to
/home/hbase/logs/hbase-root-master-localhost.localdomain.out
slaveA: starting regionserver,
logging to /home/hbase/bin/../logs/hbase-root-regionserver-localhost.
localdomain.out
masterH: starting regionserver,
 logging to /home/hbase/bin/../logs/hbase-root-regionserver-localhost.
localdomain.out
slaveB: starting regionserver,
logging to /home/hbase/bin/../logs/hbase-root-regionserver-localhost.
localdomain.out
```

启动后，分别登录 masterH、slaveA、slaveB 节点，查看各自的 HBase 进程状态，进行验证，效果如下所示，至此集群部署成功。

```
[root@masterH /]# jps
19469 NameNode
19607 DataNode
19774 SecondaryNameNode
22359 HQuorumPeer
22431 HMaster
22571 HRegionServ
23311 jps
[root@slaveA /]# jps
29486 DataNode
30480 HQuorumPeer
31358 jps
30578 HRegionServer
[root@slaveB /]# jps
27229 DataNode
27985 HQuorumPeer
28599 jps
28088 HRegionServer
```

6.5.2 自动化脚本

在集群环境下，很多数据的处理不能全靠手动输入命令，而是需要批量处理，在 HBase

中，提供了多种方式，可以实现批处理。

1. 批处理命令

可以通过建立一个文本文件，将要执行的 HBase 命令顺次写入，通过 hbase shell filename 这种方式来执行，以达到批量处理的效果。

例如，这里创建一张新表，然后在表中写入数据，并查询写入后的结果，操作效果如下所示：

```
[root@masterH /]# vi /home/bigdatabookprj/hbase_commandlist
create 'bigdata:bat_tb', 'batname'
t = get_table 'bigdata:bat_tb'
list
t.put '1', 'batname:name1', 'value1'
t.put '2', 'batname:name2', 'value2'
t.put '3', 'batname:name3', 'value3'
t.put '4', 'batname:name4', 'value4'
t.put '5', 'batname:name5', 'value5'
t.get '1'
[root@ masterH /]# hbase shell /home/bigdatabookprj/hbase_commandlist
0 row(s) in 1.6350 seconds
0 row(s) in 0.0020 seconds
TABLE
bigdata:bat_tb
bigdata:emp_sign_tb
bigdata:emp_tb
hbase_test
testhbase
5 row(s) in 0.0240 seconds

0 row(s) in 0.1830 seconds
0 row(s) in 0.0050 seconds
0 row(s) in 0.0050 seconds
0 row(s) in 0.0050 seconds
0 row(s) in 0.0080 seconds
COLUMN                                          CELL
 batname:name1                                   timestamp=1456928072609,
value=value1
1 row(s) in 0.0220 seconds
```

2. 基于 Shell 脚本

通过 echo “create ‘tablename’,‘name’” | hbase shell 这种形式，可以将 Shell 脚本与 HBase Shell 命令关联起来。

例如，这里要批量创建 10 张表，如果采用前面的批处理方法，创建命令将要写 10 遍，而这里，只通过 for 循环，就可以完成，先编写一个 Shell 脚本，hbase_shell.sh，然后执行，运行效果如下所示：

```
[root@masterH bigdatabookprj]# vi hbase_shell.sh
#!/bin/sh
for i in tb1 tb2 tb3
 do
  echo "begin create table $i"
  echo "create '$i','name'" | hbase shell
  echo "end create table $i"
 done
```

```
echo "list" | hbase shell
[root@masterH bigdatabookprj]# chmod u+x hbase_shell.sh
[root@masterH bigdatabookprj]# ./hbase_shell.sh
begin create table tb1
0 row(s) in 1.9200 seconds
Hbase::Table - tb1
end create table tb1
begin create table tb2
0 row(s) in 1.9610 seconds
Hbase::Table - tb2
end create table tb2
begin create table tb3
0 row(s) in 1.6800 seconds
Hbase::Table - tb3
end create table tb3
...
hbase_test
tb1
tb2
tb3
testhbase
8 row(s) in 1.0240 seconds
...
```

6.5.3 权限管理

HBase 集群中支持用户的授权管理，可以细粒度地针对单表、单列簇、单列进行授权，权限可细分为可读、可写、可执行、可创建、可管理，分别对应 READ('R')、WRITE('W')、EXEC('X')、CREATE('C')、ADMIN('A')，即 RWXCA。

运行 HBase Security 模块时，需要在 conf/hbase-site.xm 文件中配置以下参数，配置完毕后，重新启动 HBase：

```
<property>
    <name>hbase.security.authorization</name>
    <value>true</value>
</property>
<property>
    <name>hbase.coprocessor.master.classes</name>
    <value>
        org.apache.hadoop.hbase.security.access.AccessController
    </value>
</property>
<property>
    <name>hbase.coprocessor.region.classes</name>
    <value>
        org.apache.hadoop.hbase.security.token.TokenProvider,
        org.apache.hadoop.hbase.security.access.AccessController
    </value>
</property>
```

1. 权限分配

grant 命令用于权限分配，它可以向指定的用户，细粒度地划分权限，它的命令语法是：

```
grant <user><permissions>[<table>[<column family>[<column qualifier>]]
```

(1) 为 user1 用户分配表 tb1 的可读可写权限，操作如下：

```
hbase(main):002:0> grant 'user1','RW','tb1'
0 row(s) in 2.3390 seconds
```

(2) 为 user1 用户分配表 tb2 的全部权限，操作如下：

```
hbase(main):003:0> grant 'user1','RWXCA','tb2'
0 row(s) in 0.0960 seconds
```

(3) 为 user1 用户分配表 tb3 的 name 列簇的可读可写可执行权限，操作如下：

```
hbase(main):004:0> grant 'user1','RWX','tb3','name'
0 row(s) in 0.1240 seconds
```

2. 权限查看

user_permission 命令可以查看某张表的权限分配情况，如下所示：

```
hbase(main):007:0> user_permission 'tb1'
User
Table,Family,Qualifier:Permission
 user1                                    tb1,,: [Permission:
actions=READ,WRITE]
 root                  tb1,,: [Permission:
actions=READ,WRITE,EXEC,CREATE,ADMIN]
2 row(s) in 0.1780 seconds
```

也可以查看全部表的权限分配情况，这里要使用正则表达式*，如下所示：

```
hbase(main):008:0> user_permission '*'
User
Table,Family,Qualifier:Permission
 root   bigdata:bat_tb,,: [Permission:
actions=READ,WRITE,EXEC,CREATE,ADMIN]
 root   bigdata:emp_sign_tb,,: [Permission:
actions=READ,WRITE,EXEC,CREATE,ADMIN]
 user1  tb1,,: [Permission: actions=READ,WRITE]
 root   tb1,,: [Permission: actions=READ,WRITE,EXEC,CREATE,ADMIN]
 user1  tb2,,: [Permission: actions=READ,WRITE,EXEC,CREATE,ADMIN]
 root   tb2,,: [Permission: actions=READ,WRITE,EXEC,CREATE,ADMIN]
 user1  tb3,name,: [Permission: actions=READ,WRITE,EXEC]
 root  tb3,,: [Permission: actions=READ,WRITE,EXEC,CREATE,ADMIN]
8 row(s) in 0.3090 seconds
```

3. 权限回收

revoke 命令用于权限回收，其语法与分配权限类似：

```
revoke <user><table><column family> <column qualifier>
```

可以指定回收某个用户针对某个表、某个列簇或某个列的权限。

例如，回收 user1 用户在表 tb1 上的权限，如下所示：

```
hbase(main):012:0> revoke 'user1','tb1'
0 row(s) in 0.1090 seconds
hbase(main):013:0> user_permission 'tb1'
User
Table,Family,Qualifier:Permission
 root       tb1,,: [Permission: actions=READ,WRITE,EXEC,CREATE,ADMIN]
1 row(s) in 0.0670 seconds
```

6.5.4 集群调度

集群调度重点是资源的再分配，要经常对表进行分析，对于所涉及的 Region 要进行合理的调整，如果某些 RegionServer 服务器上的 Region 过多，可以动态迁移，将这些 Region 调整至 Region 数目较少的 RegionServer 上；对某些过大的 Region 文件，通过分裂，可以平衡负载；对一些 Region 可以归并，以对资源做更有效的利用。

1. 迁移

通过 move 命令，可以将指定的 Region 迁移至另外的 RegionServer 服务器上，它的使用语法是：

```
move 'ENCODED_REGIONNAME', 'SERVER_NAME'
```

ENCODED_REGIONNAME 是指 Region 的名称；SERVER_NAME 是指要迁至的目标服务器，通常要由主机名、端口号、时间戳三元信息组成，如 hostname, 60020, 128949312175。如果 SERVER_NAME 不指定，HBase 会自动迁移。

例如，迁移一个名为 b67a4dea80dd7c3d2495fbb0c798bfb1 的 Region 至 slaveA 中，如下所示：

```
hbase(main):005:0> move
'b67a4dea80dd7c3d2495fbb0c798bfb1','slaveA,60020,122323232323'
0 row(s) in 0.0320 seconds
```

2. 分裂

通过 split 命令进行分裂，它可以针对表进行操作，也可以通过行键对 Region 进行操作，使用语法如下：

- split ‘tableName’。
- split ‘regionName’, ‘splitKey’。

这里以 emp_sign_tb 为例，对其进行分裂，如下所示：

```
hbase(main):013:0> split 'bigdata:emp_sign_tb'
0 row(s) in 0.0480 seconds
```

3. 归并

major_compact 可对于一张表进行数据归并，以节约资源，通常，在管理时，可以在业务压力较少的晚上进行手动归并，可以先针对较大的表进行归并，也可以针对全部的表进行定期归并，它的使用语法是：major_compact ‘tbname’。要注意的是，在使用本命令时，最好关闭 Region 负载平衡功能(balance_switch 命令后跟 false 表示执行关闭，跟 true 表示执行开启)。

例如，对表 tb1 进行归并，归并前，先关闭自动负载均衡，运行完毕后，再开启，操作如下所示：

```
hbase(main):017:0> balance_switch false
false
0 row(s) in 0.0130 seconds
hbase(main):018:0> major_compact 'tb1'
```

```
0 row(s) in 0.0450 seconds
hbase(main):019:0> balancer
false
0 row(s) in 0.0270 seconds
hbase(main):020:0> balance_switch true
false
0 row(s) in 0.0140 seconds
hbase(main):021:0> balancer
true
0 row(s) in 0.0100 seconds
```

4. RegionServer 节点的管理

除了对 Region 调整之外，在很多时候，也要关注 RegionServer 资源，当整个集群负载较大时，要动态增加 RegionServer；某些 RegionServer 发生故障时，要及时下线故障节点。

(1) 增加 RegionServer 节点的流程如下。

① 在 Hadoop 集群中增加 DataNode 节点，并开通成功。

② 在 HMaster 的 RegionServer 配置文件中添加新的 DataNode 节点。

③ 对该节点推送 HBase 运行文件及配置文件。

④ 在新节点上的 hbase/bin 目录中，运行 hbase-daemon.sh start regionserver，执行新节点的运行。

(2) 对于故障节点的下线流程则是这样的。

① 关闭集群负载均衡。

② 在主节点的 hbase/bin 目录中执行 graceful_stop.sh HOSTNAME，其中 HOSTNAME 是指要关闭的节点主机名。

③ 开启集群负载均衡。

④ 在故障节点上执行 hbase-daemon.sh stop regionserver。

5. Region 的检查与修复

HBase 中提供了 hbck 命令，用于检查与修复 Region，通过 hbase hbck 可以直接对集群中的 Region 进行检查：

```
[root@masterH bin]# hbase hbck
...
Summary:
  hbase:meta is okay.
    Number of regions: 1
    Deployed on:  masterH,60020,1456904989052
  hbase:acl is okay.
    Number of regions: 1
    Deployed on:  masterH,60020,1456904989052
  tbl is okay.
    Number of regions: 1
    Deployed on:  masterH,60020,1456904989052
  ...
0 inconsistencies detected.
Status: OK
...
```

通过输入 hbase hbck -fix，可自动地对问题的 Region 进行修复，当然，对于修复存在问题的，可以配合通过日志信息查看故障的原因。

6.5.5 日志分析

在集群管理中，不可避免地要处理各类故障，分析并定位故障最好的方式，是查看日志，在 HBase 中，提供了 Master、RegionServer 等不同粒度的日志文件，通过查看日志，可以细粒度地定位各类问题。

在 hbase/logs 目录中，有三类日志文件。

- HMaster 运行日志：以 hbase-root-master-hostname.log 文件名存放。
- RegionServer 运行日志：以 hbase-root-regionserver-hostname.log 文件名存放。
- ZooKeeper 运行日志，以 hbase-root-zookeeper-hostname.log 文件名存放。

通过“tail -f 日志文件”，可以查看日志文件，如下所示：

```
[root@masterH logs]# tail -f hbase-root-master-iZ23d4by1laZ.log
2016-01-21 15:29:17,805 INFO
master.ReplicationLogCleaner: Stopping
replicationLogCleaner-0x151ebb359030003, quorum=iZ23d4by1laZ:2181,
baseZNode=/hbase
2016-01-21 15:29:17,809 INFO  [RpcServer.responder] ipc.RpcServer:
RpcServer.responder: stopped
2016-01-21 15:29:17,809 INFO  [RpcServer.responder] ipc.RpcServer:
RpcServer.responder: stopping
...
```

在具体的日志文件中，有多种日志告警级别，分别为 DEBUG、INFO、WARN，在运行调试期间，可以开启 DEBUG，但在集群正式运行时，往往要关闭 DEBUG 级日志输出，调整不同级别的日志输出通过 hbase/conf 目录中的 log4j.properties 文件来进行：

```
[root@masterH conf]# cat log4j.properties
# Custom Logging levels

log4j.logger.org.apache.zookeeper=INFO
#log4j.logger.org.apache.hadoop.fs.FSNamesystem=DEBUG
log4j.logger.org.apache.hadoop.hbase=DEBUG
# Make these two classes INFO-level. Make them DEBUG to see more zk debug.
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZKUtil=INFO
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZooKeeperWatcher=INFO
#log4j.logger.org.apache.hadoop.dfs=DEBUG
# Set this class to log INFO only otherwise its OTT
# Enable this to get detailed connection error/retry logging.
#
log4j.logger.org.apache.hadoop.hbase.client.HConnectionManager$HConnecti
onImplementation=TRACE
...
```

通过修改上面标记的参数，如将 DEBUG 改为 INFO，即可以只输出 INFO 级以上日志信息，在集群中修改本文件后，要及时同步到其他节点，并重新启动 HBase 集群生效。

在不少规模较大的集群中，需要建立统一的日志服务器，进行统一的 HBase 日志的收集、定位、分析、决策。

6.6　小　　结

本章围绕 HBase 的基本概念、基本原理、数据模型、伪分布式安装部署、Shell 命令、HBase 集群部署与管理进行了介绍与实践。

通过本章的学习，读者应能够基本掌握 HBase 的部署与使用，为后续学习中做更大规模的集群实践提供支撑。

第 7 章

HBase 编程开发

学习目标

本章将详细介绍 HBase 编程框架、API 接口、表编程、数据编程、集群编程等。通过本章的学习，读者将能够深入掌握 HBase 的二次开发与集群管理，为未来在项目中灵活运用 HBase 打好基础。

本章要点

- HBase 编程框架
- HBase rest 接口
- HBase API 接口
- HBase 编程示例

7.1　HBase 的编程接口

HBase 提供了多种编程接口，分别满足不同的用户场景，这些接口包括：rest、thrift、avro、Java 原生 API 等方式。其中，基于 rest 的接口支持 HTTP 的文本数据互操作，基于 thrift 即支持文本，也支持二进制流；基于 avro 的方式只支持二进制流，图 7-1 给出了编程接口的示意图。

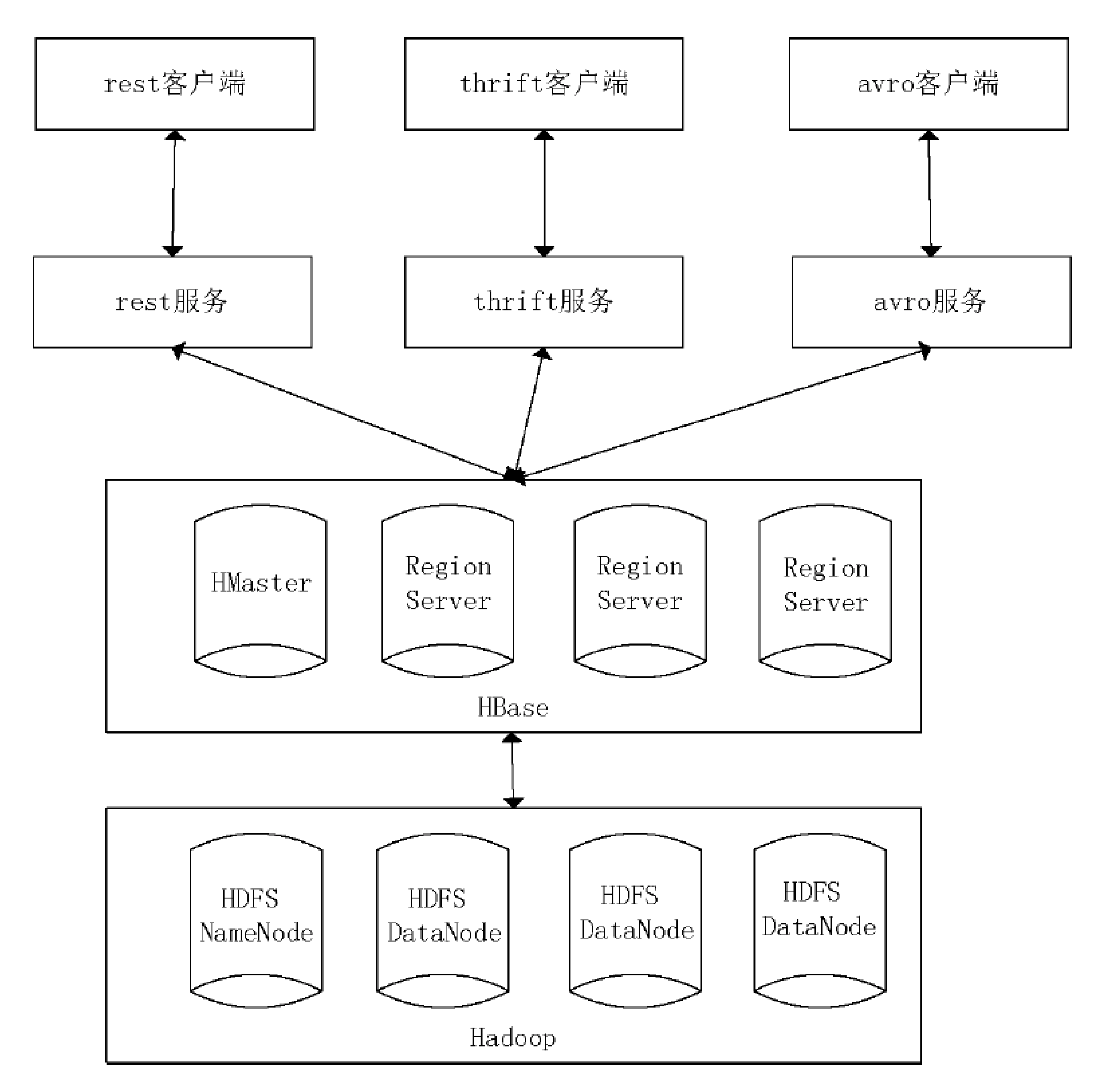


图 7-1　HBase 编程接口

图 7-1 中，针对每一类接口，在 HBase 中都会启动对应的服务作为支撑，通过服务层的引入，来实现不同用户的跨平台访问需求，

7.1.1　rest 编程接口

rest 服务基于 HTTP 协议，在 HBase 中内置了 rest 服务，通过 hbase rest starts，可以直接启动 rest 服务，启动成功后，默认 rest 服务运行在前台，默认端口为 8080；也可以通过在命令后面加入-p 参数，来更改默认端口号。如将默认端口改成 8090：hbase rest start -p 8090，也能够调用 hbase/bin 目录下的 hbase-daemon.sh 脚本，以 hbase-daemon.sh start rest 的方式完成 rest 服务的后台启动。

1. rest 服务启动

下面以后台的方式启动 rest 服务，默认端口改为 8090。启动完毕之后，通过查看 Linux 绑定的端口，判断服务是否启动成功。

```
[root@www bin]# hbase-daemon.sh start rest -p 8090
starting rest, logging to /alidata/hbase/logs/hbase-root-rest-www.out
[root@www bin]# netstat -natp | grep 8090
tcp        0      0 0.0.0.0:8090              0.0.0.0:*
LISTEN      9389/java
[root@www bin]#
```

2. rest 服务停止

若 rest 在前台启动，可以通过 Ctrl+C 的方式进行关闭，也可以通过 ps -ef 去查找存活的进程，以借助 kill 关闭服务进程的方式关闭；若 rest 是后台运行，可以通过 hbase-daemon.sh stop rest 来完成服务的关闭。这里关闭 rest 服务，如下所示，需要注意的是，如果是指定端口号启动的 rest，在关闭时，同样要指定一下 rest 当前运行的端口。

```
[root@www bin]# hbase-daemon.sh stop rest -p 8090
stopping rest..
[root@www bin]# netstat -natp | grep 8090
[root@www bin]#
```

3. rest 服务连接

rest 是基于 HTTP 协议运行的，针对 HBase 数据库的增删改查，分别对应于 HTTP 协议中的相应请求方法，表 7-1 给出了对应的关系。

表 7-1　rest 请求对应的关系

rest 功能	HTTP 方法	备　注
CREATE	PUT	创建表等
UPDATE	POST/PUT	POST 用于修改，PUT 用于替换
DELETE	DELETE	删除表、数据等
READ	GET	查看集群状态、数据表信息等

在浏览器中直接输入 URL 来查看结果(调用 GET 方法)，如图 7-2 所示。也可以通过 Linux 的 wget 或 curl 命令查看服务结果。



图 7-2　HBase 编程接口

http://hostname:port/这种方式会返回当前所有的 HBase 表名，可以通过参数指定返回的信息格式。

例如，以文本的形式返回，以 curl 命令为例：

```
[root@www bin]# curl -H "Accept: text/plain" http://iz23d4by1laz:8090/
bigdata:bat_tb
bigdata:emp_sign_tb
bigdata:emp_tb
exam_tb
hbase_test
tb1
tb2
tb3
testhbase
```

以 XML 格式返回，如下所示：

```
[root@www bin]# curl -H "Accept: text/xml" http://iz23d4by1laz:8090/
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<TableList>
<table name="bigdata:bat_tb"/>
<table name="bigdata:emp_sign_tb"/>
<table name="bigdata:emp_tb"/>
<table name="exam_tb"/>
<table name="hbase_test"/>
<table name="tb1"/>
<table name="tb2"/>
<table name="tb3"/>
<table name="testhbase"/>
</TableList>
```

4. 查看表结构

在 URL 链接后面跟表名，跟 schema 关键字，如 http://hostname:port/tablename/schema，可以查看指定表的结构。这里查看表 hbase_test 的结构，如下所示：

```
[root@www hbase]# curl -H "Accept: text/plain"
http://iz23d4by1laz:8090/hbase_test/schema
{ NAME=> 'hbase_test', IS_META => 'false', COLUMNS => [ { NAME => 'f1',
BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false',
KEEP_DELETED_CELLS => 'false', DATA_BLOCK_ENCODING => 'NONE', COMPRESSION
=> 'NONE', TTL => '2147483647', MIN_VERSIONS => '0', BLOCKCACHE => 'true',
BLOCKSIZE => '65536', REPLICATION_SCOPE => '0' }, { NAME => 'f2', BLOOMFILTER
=> 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'false',
DATA_BLOCK_ENCODING => 'NONE', COMPRESSION => 'NONE', TTL => '2147483647',
MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536',
REPLICATION_SCOPE => '0' } ] }
```

5. 删除表

在 URL 中构造 delete 请求，通过 detete /tablename/schema，可以实现对表的删除。例如，当前删除 tb1 表，操作如下：

```
[root@www lib]# curl -v -X delete http://iz23d4by1laz:8090/tb1/schema
* About to connect() to www.rdesec.com port 8090
*  Trying 121.40.126.204... connected
* Connected to www.rdesec.com (121.40.126.204) port 8090
> delete /tb1/schema HTTP/1.1
```

```
> User-Agent: curl/7.15.5 (x86_64-redhat-linux-gnu) libcurl/7.15.5
OpenSSL/0.9.8b zlib/1.2.3 libidn/0.6.5
> Host: www.rdesec.com:8090
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 0
* Connection #0 to host www.rdesec.com left intact
* Closing connection #0
```

这里为了方便读者查看，使用了 curl 的-v 参数，可以打印 HTTP 交互的细节，使用-X 参数用于构造请求方法。执行完毕后，通过 http://www.rdesec.com:8090/tb1/exists 的方法查询是否已删除：

```
[root@www lib]# curl http:// iz23d4by1laz:8090/tb1/exists
Not found
```

6. 查看数据

在 URL 中输入“http://hostname:port/tablename/rowkey/”，可以查看指定行键值的列数据。这里查询表 hbase_test 行键值为 1 的列数据，如下所示：

```
[root@www hbase]# curl http://iz23d4by1laz:8090/hbase_test/1/
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<CellSet>
<Row key="MQ==">
<Cell column="ZjE6" timestamp="1456718706841">YQ==</Cell>
<Cell column="ZjI6" timestamp="1456718706938">YTE=</Cell>
</Row>
</CellSet>
```

返回的值以 base64 的形式进行编码，在 Linux 中通过 base64 命令可进行解码，如上面的列字段名 ZjE6，经解码后为 f1，如下所示：

```
[root@www hbase]# echo ZjE6 | base64 -d
f1
```

7. 提交数据

通过构造 PUT 或 POST 方法的 URL，可以实现表的创建、列数据的增加，这里以数据新增为例，URL 的类型为 http://hostname:port/tablename/rowkey/<column>(‘:’<qualifier>)，提交的为 PUT 方法，将要提交的列数据保存到文本文件中。

当前创建一个文本文件 tt，写入数据 test，向 hbase_test 表的 f1:status 字段进行数据提交，行键值为 8，操作如下所示：

```
[root@www hjava]# cat tt
test
[root@www hjava]# curl -v -T "/home/bigdatabookprj/hjava/tt"
        http:// iz23d4by1laz:8090/hbase_test/8/f1:status
* About to connect() to www.rdesec.com port 8090
*   Trying 121.40.126.204... connected
* Connected to www.rdesec.com (121.40.126.204) port 8090
> PUT /hbase_test/8/f1:status HTTP/1.1
> User-Agent: curl/7.15.5 (x86_64-redhat-linux-gnu) libcurl/7.15.5
OpenSSL/0.9.8b zlib/1.2.3 libidn/0.6.5
> Host: www.rdesec.com:8090
```

```
> Accept: */*
> Content-Length: 5
> Expect: 100-continue
>
< HTTP/1.1 100 Continue
< HTTP/1.1 200 OK
< Content-Length: 0
* Connection #0 to host www.rdesec.com left intact
* Closing connection #0
[root@www hjava]# curl http://www.rdesec.com:8090/hbase_test/8/f1:status
test
```

8. 查看表元数据

在 URL 链接后面跟表名，跟 regions 关键字，如 http://hostname:port/tablename/regions，可以查看指定表的元数据信息。例如：

```
[root@www ~]# curl -H "Accept: text/xml" http://
iz23d4by1laz:8090/hbase_test/regions
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<TableInfo name="hbase_test">
<Region endKey="" id="1456718705230" location="iZ23d4by1laZ:60020"
name="hbase_test,,1456718705230.e677c3827cf46503e5a23c663cd14b92."
startKey=""/>
</TableInfo>
```

9. 查看集群状态

在 URL 链接后面跟/status/cluster，可以查看集群的状态。

如 http://hostname:port//status/cluster。例如：

```
[root@www ~]# curl http://iz23d4by1laz:8090/status/cluster
1 live servers, 0 dead servers, 12.0000 average load
1 live servers
    iZ23d4by1laZ:60020 1457240254015
        requests=0, regions=12
...
         testhbase,,1453080469131.0c3da9d154163bbb2b106ae878acc9cc.
            stores=2
            storefiless=2
            storefileSizeMB=0
            memstoreSizeMB=0
            storefileIndexSizeMB=0
            readRequestsCount=0
            writeRequestsCount=0
            rootIndexSizeKB=0
            totalStaticIndexSizeKB=0
            totalStaticBloomSizeKB=0
            totalCompactingKVs=0
            currentCompactedKVs=0
```

10. 编程接口

可以基于多种编程语言开发 HBase rest 应用程序，在 HBase 中，提供了基于 Java 语言的 rest 接口包。下面以 Java 语言为例，编写一个 rest 连接程序，用于查询 hbase_test 表中的某一键值的信息。源代码如下所示：

```
import java.util.Iterator;
```

```
import java.util.List;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.rest.client.Client;
import org.apache.hadoop.hbase.rest.client.Cluster;
import org.apache.hadoop.hbase.rest.client.RemoteHTable;

public class restexam {

    public static void main(String[] args) throws Exception {
        //初始化并连接 HBase 集群
        Cluster hbasecluster = new Cluster();
        hbasecluster.add("iZ23d4by1laZ", 8080);
        Client restclient = new Client(hbasecluster);
        //定位要操作的 HBase 数据表
        RemoteHTable hbasetable =
          new RemoteHTable(restclient, "hbase_test");
        //设定行键值
        Get rowkey = new Get(Bytes.toBytes("7"));
        rowkey.addFamily(Bytes.toBytes("f2"));
        //返回结果集合
        Result res = hbasetable.get(rowkey);
        List<Cell> celist = res.listCells();
        //迭代结果集，并查看结果
        for(Iterator i=celist.iterator(); i.hasNext(); )
        {
            Cell ce = (Cell)i.next();
            System.out.println(
              "descripinfor: " + ce + " value: " + new String(ce.getValue()));
        }
        hbasetable.close();
    }
}
```

在编译时，需要引入 hbase-client-0.98.3-hadoop1.jar、hbase-client-0.98.3-hadoop1.jar 两个包，由于编译环境已配置 lib 环境变量并指向 hbase/lib，因此可以直接编译，编译与运行的效果如下所示：

```
[root@www hjava]# javac restexam.java
[root@www hjava]# java restexam
descripinfor: 7/f1:/1458390524322/Put/vlen=0/mvcc=0 value:
descripinfor: 7/f1:status/1458398719648/Put/vlen=4/mvcc=0 value: liu
descripinfor: 7/f1:status1/1458399002174/Put/vlen=0/mvcc=0 value:
```

小总结

关于 rest 接口。

① 基于 HTTP 的 rest 接口具有松耦合、跨平台、与编程语言无关性的特点，能够方便各类编程语言使用。

② 在编程语言中使用 rest，应借助于网络 socket 接口对 HTTP 进行封装，即可以完成与 HBase 的交互。

7.1.2 thrift 接口

thrift 是 HBase 提供的一种跨平台 RPC 服务，可以支持用 C++、PHP、Python、Java 等多种语言开发 HBase thrift 应用程序。启停 thrift 服务与 rest 相类似，不同之处是，服务名称为 thrift 或者 thrift2，如 hbase thrift start，默认的启动端口为 9090。

目前，开发 thrift 应用程序多采用 thrift2 作为网关服务，相比于 thrift 服务，thrift2 有较大的改进，启动 thrift2 的示例如下：

```
[root@www ~]# hbase thrift2 start
...
2016-03-22 09:51:04,732 INFO  [main] http.HttpServer: Jetty bound to port
9095
...
2016-03-22 09:51:05,411 INFO  [main] thrift2.ThriftServer: starting HBase
ThreadPool Thrift server on 0.0.0.0/0.0.0.0:9090
```

默认的 9095 是 thrift 的 Web 管理界面，效果如图 7-3 所示。

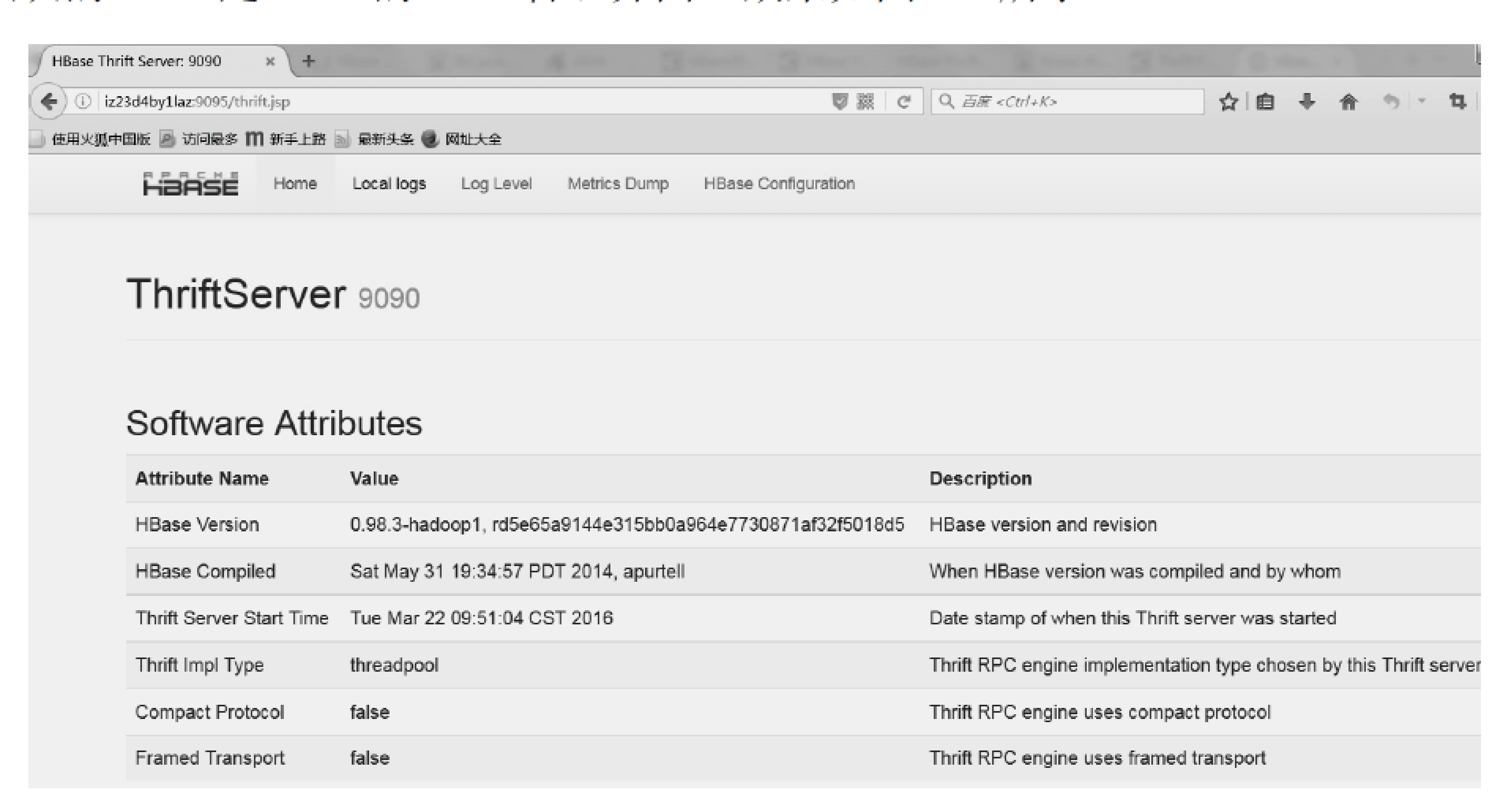


图 7-3 HBase 的 thrift 管理界面

9090 是接口服务端口，thrift 提供一整套的 API 接口类库与方法、函数，这里以 Java 语言为例，开发了一个简单的数据提交与查询系统，针对 hbase_test 表，提交一条记录，并查询记录集。

源代码如下所示：

```
import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.List;
import org.apache.hadoop.hbase.thrift2.generated.TColumnValue;
import org.apache.hadoop.hbase.thrift2.generated.TGet;
import org.apache.hadoop.hbase.thrift2.generated.THBaseService;
import org.apache.hadoop.hbase.thrift2.generated.TIOError;
import org.apache.hadoop.hbase.thrift2.generated.TPut;
import org.apache.hadoop.hbase.thrift2.generated.TResult;
import org.apache.thrift.TException;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TProtocol;
```

```
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;

public class thrifttest {
    public static void main(String[] args) throws TIOError, TException {
        String hostname = "iZ23d4by1laZ";
        int port = 9090;
        int timeout = 10000;
        //构建 thrift 连接，并获取连接
        TTransport thriftsocket = new TSocket(hostname, port, timeout);
        THBaseService.Iface thriftclient = new THBaseService.Client(
          new TBinaryProtocol(thriftsocket));
        thriftsocket.open();

        //构造数据提交条件
        ByteBuffer hbasetable = ByteBuffer.wrap("hbase_test".getBytes());
        TPut rowkey = new TPut();
        //行键值为 9，字段为 f1:status，值为 run
        rowkey.setRow("9".getBytes());
        TColumnValue columnf1 = new TColumnValue();
        columnf1.setFamily("f1".getBytes());
        columnf1.setQualifier("status,".getBytes());
        columnf1.setValue("run".getBytes());

        //行键值为 9，字段为 f2:status，值为 stop
        TColumnValue columnf2 = new TColumnValue();
        columnf2.setFamily("f2".getBytes());
        columnf2.setQualifier("status,".getBytes());
        columnf2.setValue("stop".getBytes());

        //提交记录
        List<TColumnValue> columnlist = new ArrayList<TColumnValue>();
        columnlist.add(columnf1);
        columnlist.add(columnf2);
        rowkey.setColumnValues(columnlist);
        thriftclient.put(hbasetable, rowkey);

        //获取提交的行键值 9 的结果
        TGet rowget = new TGet();
        rowget.setRow("9".getBytes());
        TResult res = thriftclient.get(hbasetable, rowget);
        for (TColumnValue reskey : res.getColumnValues()) {
            System.out.print("key : " + new String(reskey.getFamily())
              + ":" + new String(reskey.getQualifier()));
            System.out.print("value : " + new String(reskey.getValue()));
            System.out.print(",timestamp : " + reskey.getTimestamp() + "\n");
        }

        thriftsocket.close();
    }
}
```

在编译 thrift 应用程序前，需要加载 hbase-thrift-0.98.3-hadoop1.jar 包，编译运行结果如下所示：

```
[root@www hjava]# javac thrifttest.java
[root@www hjava]# java thrifttest
key : f1:status,value : run,timestamp : 1458615999886
key : f2:status,value : stop,timestamp : 1458615999886
```

通过 HBase Shell 验证提交的数据，如下所示：

```
hbase(main):007:0> get 'hbase_test','9'
COLUMN                          CELL
 f1:status,                      timestamp=1458615999886, value=run
 f2:status,                      timestamp=1458615999886, value=stop
2 row(s) in 0.0130 seconds
```

7.1.3 Java API 接口

HBase 本身是基于 Java 开发的，因此，也提供了一整套的 Java API 开发接口，整个接口方法非常完善，包括命令空间管理、表级管理、列簇级管理、数据(增删改查、导入、导出)、集群调度、状态监测、集群优化等。

HBase 官方网站(http://hbase.apache.org/apidocs/overview-summary.html)提供了上述包、类、方法的详细使用说明，用于查询所有的 HBase Java 包以及使用方法，如图 7-4 所示。

Overview Package Class Use Tree Deprecated Index Help

Prev Next Frames No Frames All Classes

Apache HBase 2.0.0-SNAPSHOT API

Packages

Package	Description
com.google.protobuf	
org.apache.hadoop.hbase	
org.apache.hadoop.hbase.classification	
org.apache.hadoop.hbase.classification.tools	
org.apache.hadoop.hbase.client	Provides HBase Client
org.apache.hadoop.hbase.client.backoff	
org.apache.hadoop.hbase.client.example	
org.apache.hadoop.hbase.client.replication	
org.apache.hadoop.hbase.client.security	
org.apache.hadoop.hbase.conf	
org.apache.hadoop.hbase.coordination	
org.apache.hadoop.hbase.coprocessor	Table of Contents
org.apache.hadoop.hbase.errorhandling	
org.apache.hadoop.hbase.exceptions	
org.apache.hadoop.hbase.filter	Provides row-level filters applied to HRegion scan results during calls to ResultScanner.next().
org.apache.hadoop.hbase.http	Copied from hadoop source code. See https://issues.apache.org/jira/browse/HADOOP-10232 to know why.
org.apache.hadoop.hbase.http.conf	
org.apache.hadoop.hbase.http.jmx	This package provides access to JMX primarily through the JMXJsonServlet class.
org.apache.hadoop.hbase.http.log	

图 7-4　HBase 的包库列表

编程流程

基于 Java API 进行 HBase 编程，需要遵循初始化连接 HBase、执行相关表与数据操作、编译与运行三个步骤。

首先，建立初始化连接。需要通过 Configuration 类建立与 Zookeeper 的连接通道，需要指定 HBase 的 ZooKeeper IP 与端口号，默认为 2181。

在初始化连接代码中，需要先导入 org.apache.hadoop.conf.Configuration 包。

具体代码如下所示：

```
import org.apache.hadoop.conf.Configuration;
...
private static Configuration hbaseconn = null;
static {
    Configuration connconf = new Configuration();
    connconf.set("hbase.zookeeper.quorum", "127.0.0.1");
    connconf.set("hbase.zookeeper.property.clientPort", "2181");
    hbaseconn = HBaseConfiguration.create(connconf);
}
```

其次，利用 HBase Java API 各类接口完成所需要的表、数据、集群管理等操作，这些会在后面的内容中详细介绍其使用。

最后，对代码进行编译与运行。编译时，需要指定大量的 HBase 库，这些库位于 hbase/lib 目录中，如图 7-5 所示。编译完毕后，可以借助 Hadoop 命令加载至集群中运行，也可以利用 java 命令独立运行。

```
[root@www hbase]# ls lib/
activation-1.1.jar                    hbase-hadoop1-compat-0.98.3-hadoop1.jar  jersey-core-1.8.jar
asm-3.1.jar                           hbase-hadoop-compat-0.98.3-hadoop1.jar   jersey-json-1.8.jar
commons-beanutils-1.7.0.jar           hbase-it-0.98.3-hadoop1.jar              jersey-server-1.8.jar
commons-beanutils-core-1.8.0.jar      hbase-it-0.98.3-hadoop1-tests.jar        jettison-1.3.1.jar
commons-cli-1.2.jar                   hbase-prefix-tree-0.98.3-hadoop1.jar     jetty-6.1.26.jar
commons-codec-1.7.jar                 hbase-protocol-0.98.3-hadoop1.jar        jetty-sslengine-6.1.26.jar
commons-collections-3.2.1.jar         hbase-server-0.98.3-hadoop1.jar          jetty-util-6.1.26.jar
commons-configuration-1.6.jar         hbase-server-0.98.3-hadoop1-tests.jar    jruby-complete-1.6.8.jar
commons-digester-1.8.jar              hbase-shell-0.98.3-hadoop1.jar           jsp-2.1-6.1.14.jar
commons-el-1.0.jar                    hbase-testing-util-0.98.3-hadoop1.jar    jsp-api-2.1-6.1.14.jar
commons-httpclient-3.1.jar            hbase-thrift-0.98.3-hadoop1.jar          jsr305-1.3.9.jar
commons-io-2.4.jar                    high-scale-lib-1.1.1.jar                 junit-4.11.jar
commons-lang-2.6.jar                  htrace-core-2.04.jar                     libthrift-0.9.0.jar
commons-logging-1.1.1.jar             httpclient-4.1.3.jar                     log4j-1.2.17.jar
commons-math-2.1.jar                  httpcore-4.1.3.jar                       metrics-core-2.1.2.jar
commons-net-1.4.1.jar                 jackson-core-asl-1.8.8.jar               netty-3.6.6.Final.jar
findbugs-annotations-1.3.9-1.jar      jackson-jaxrs-1.8.8.jar                  protobuf-java-2.5.0.jar
guava-12.0.1.jar                      jackson-mapper-asl-1.8.8.jar             ruby
hadoop-core-1.2.1.jar                 jackson-xc-1.8.8.jar                     servlet-api-2.5-6.1.14.jar
hamcrest-core-1.3.jar                 jamon-runtime-2.3.1.jar                  slf4j-api-1.6.4.jar
hbase-client-0.98.3-hadoop1.jar       jasper-compiler-5.5.23.jar               slf4j-log4j12-1.6.4.jar
hbase-common-0.98.3-hadoop1.jar       jasper-runtime-5.5.23.jar                xmlenc-0.52.jar
hbase-common-0.98.3-hadoop1-tests.jar jaxb-api-2.2.2.jar                       zookeeper-3.4.6.jar
hbase-examples-0.98.3-hadoop1.jar     jaxb-impl-2.2.3-1.jar
```

图 7-5　所需要的类库

7.1.4　Java API 示例

这里编写一个 HBase 示例，创建一张考试成绩表 exam_tb，含有一个字段 math，用于标记数学成绩，学生姓名代表行键值，向表中写入一条学生成绩信息。

1. 程序代码

根据上述需求，建立 hbaseexam.java 文件，并建立一个名为 hbaseexam 的主类，程序的代码如下：

```
import java.io.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;
```

```
public class hbaseexam {

    private static Configuration hbaseconf = null;
    static {
        Configuration testconf = new Configuration();
        testconf.set("hbase.zookeeper.quorum", "iZ23d4by1laZ");
        testconf.set("hbase.zookeeper.property.clientPort", "2181");
        hbaseconf = HBaseConfiguration.create(testconf);
    }
    public static void main (String [] agrs) {
        try {
            String strTableName = "exam_tb";
            //表创建
            HBaseAdmin hbTable = new HBaseAdmin(hbaseconf);
            HTableDescriptor tbdesc = new HTableDescriptor(strTableName);
            tbdesc.addFamily(new HColumnDescriptor("math"));
            hbTable.createTable(tbdesc);
            System.out.println("Status: [Sucess] Create Table  ok...");

            //添加记录
            HTable table = new HTable(hbaseconf, strTableName);
            Put put = new Put(Bytes.toBytes("tom"));
            put.add(Bytes.toBytes("math"),
              Bytes.toBytes(""), Bytes.toBytes("80"));
            table.put(put);
            System.out.println("Status: [Sucess] add recored  ok...");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2. 手动编译与运行

可以手动通过 javac 命令编译该 Java 文件，前提是，在编译的机器上，CLASSPATH 已设置到 hbase/lib，编译成功后，通过 java 命令运行。

如下所示：

```
[root@www hjava]# javac hbaseexam.java
[root@www hjava]# java hbaseexam
```

3. 手动加载至 Hadoop 集群运行

可以将程序打包成 JAR 文件，然后加载至 Hadoop 集群中运行，打包 JAR 的方式，是首先创建一个 manifest.mf 文件，利用 jar 命令进行打包，最后通过 hadoop jar 命令运行。

操作如下所示：

```
[root@ www hjava]# vi manifest.mf
Manifest-Version: 1.0
Main-Class: hbaseexam
[root@www hjava]# jar -cvf hbaseexam.jar hbaseexam.class
added manifest
adding: hbaseexam.class(in = 2189) (out= 1082)(deflated 50%)
...
[root@www hjava]# ls
hbaseexam.class  hbaseexam.jar  hbaseexam.java  manifest.mf
```

将生成的 hbaseexam.jar 文件加载至 Hadoop 集群中，以任务的形式执行。加载的命令如下所示：

```
[root@www hjava]# hadoop jar hbaseexam.jar hbaseexam
```

4. Eclipse 编译与运行

通过 Eclipse 进行 Java 编程开发，无疑是最为方便的，这里，先在 Eclipse 中创建一个 hbaseTest 工程，将 HBase 需要的 Java 外部包导入到工程中，导入后的效果如图 7-6 所示。

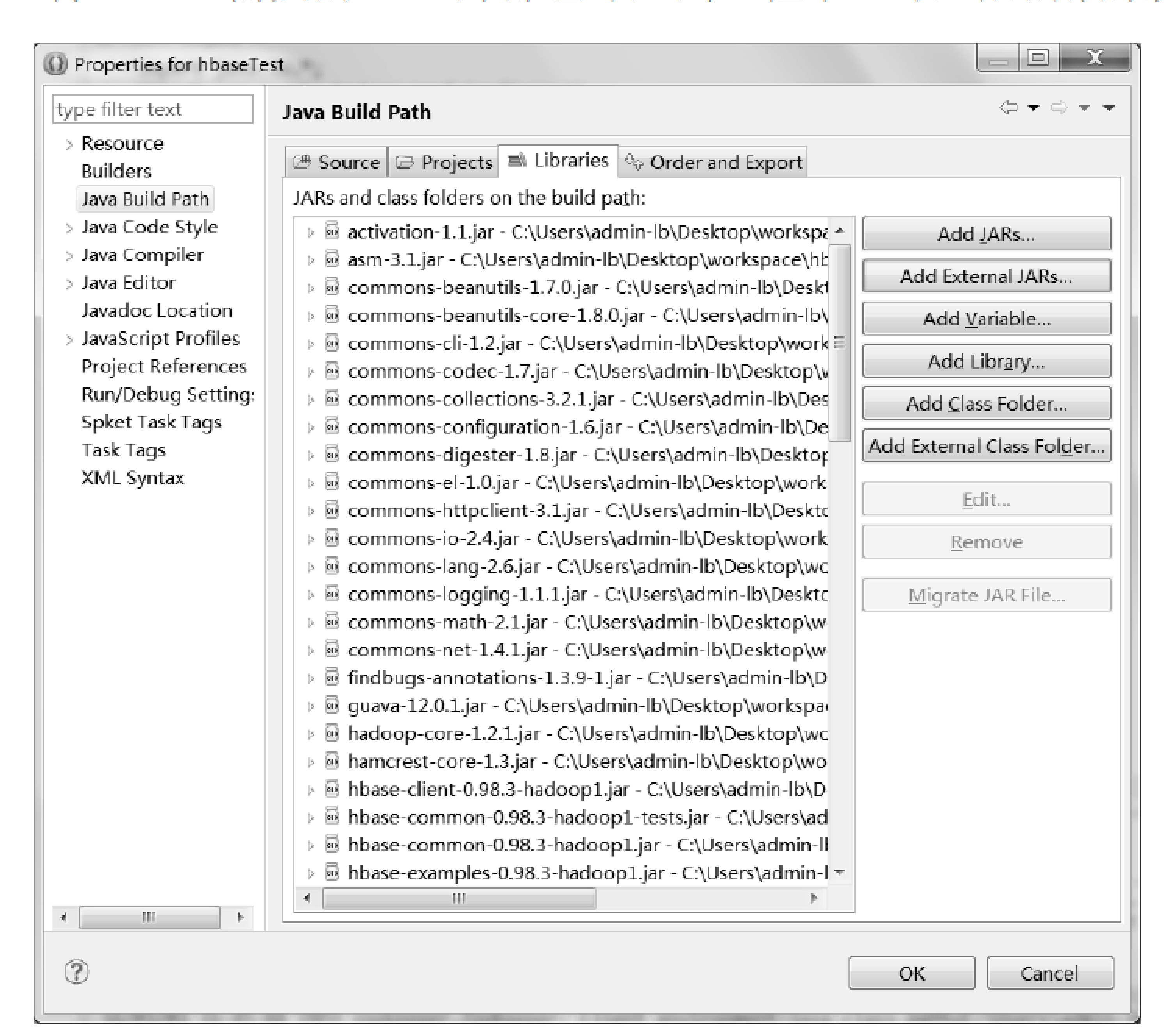


图 7-6　导入 HBase 外部包

在本工程中新建一个名为 hbaseexam.java 的文件，效果如图 7-7 所示。

Package Explorer
hbaseTest
src
(default package)
hbaseexam.java
HBaseTest.java
JRE System Library [JavaSE-1.7]
Referenced Libraries
jar

hbaseexam.java

```java
import java.io.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;

public class hbaseexam {

    private static Configuration hbaseconf = null;
    static {
        Configuration testconf = new Configuration();
        testconf.set("hbase.zookeeper.quorum", "iZ23d4by1laZ");
        testconf.set("hbase.zookeeper.property.clientPort", "2181");
        hbaseconf = HBaseConfiguration.create(testconf);
    }
    public static void  main (String [] agrs) {
        try {
            String strTableName = "exam_tb";
            // 表创建
            HBaseAdmin hbTable = new HBaseAdmin(hbaseconf);
            HTableDescriptor tbdesc = new HTableDescriptor(strTableName);
            tbdesc.addFamily(new HColumnDescriptor("math"));
            hbTable.createTable(tbdesc);
            System.out.println("Status: [Sucess] Create Table  ok...");

             // 增加记录
```

图 7-7　源码的效果

在 Eclipse 中运行本程序，可以查看执行效果，如图 7-8 所示。

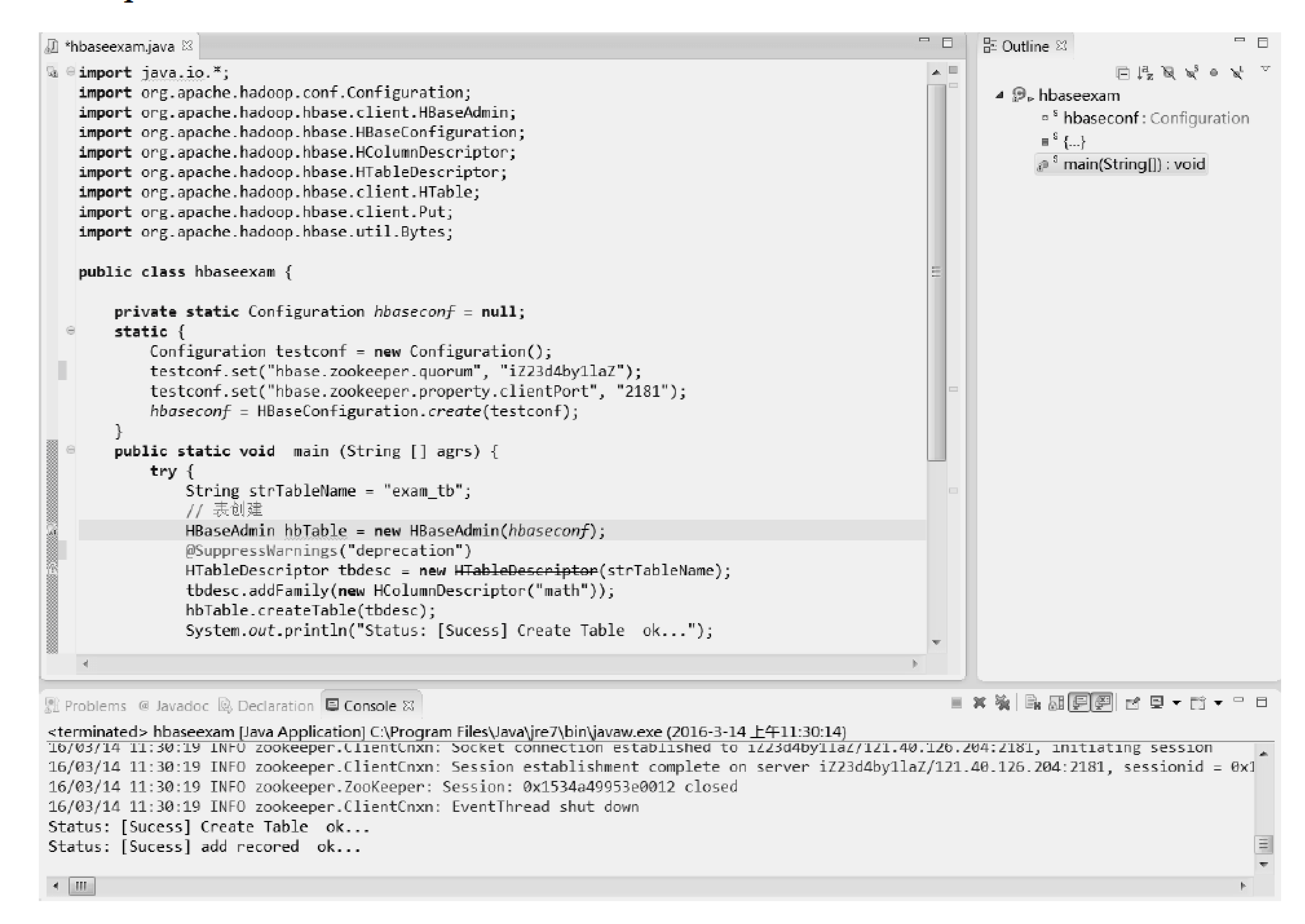


图 7-8 执行效果

5. 验证与测试

根据上面的运行效果，这里通过 HBase Shell 验证程序执行的效果，如，查看记录的创建情况，如下所示：

```
hbase(main):005:0> scan 'exam_tb'
ROW                          COLUMN+CELL
 tom                         column=math:, timestamp=1457926074892, value=80
1 row(s) in 0.0850 seconds
```

7.2 表与命名空间的编程

本节重点介绍 HBaseAdmin 类，基于这个类实现对命名空间的创建、删除，以及表的创建、删除、修改、启用、禁用等管理操作。

HBaseAdmin 在使用时需要先初始化，即先创建好 HBase 的 Configuration 类，并将该类以实例后的对象为参数传入 HBaseAdmin 构造函数，这样，就可以完成对 HBase 集群的管理了，方法原型为：

```
HBaseAdmin(Configuration)
```

具体使用代码如前面 7.1.4 小节的示例中所示：

```
import org.apache.hadoop.conf.Configuration;
...
private static Configuration hbaseconf = null;
```

```
...
HBaseAdmin admin = new HBaseAdmin(hbaseconf);
```

7.2.1　表的查看

HBaseAdmin 中提供了非常丰富的表操作的方法。

以表信息查看为例，利用 getTableNames()方法，能够查看当前集群中的所用表，并且可以在参数中输入正则表达式，用于显示满足过滤条件的表；利用 getTableDescriptor()方法可以以表名为参数，查看某张表的具体表结构描述信息(HTableDescriptor)，当然，也可以指定一个表集合为参数，查看一组表的描述信息。

类 HTableDescriptor 是数据表信息描述类，它可以存储表列簇、属性信息，支持增加、删除、查看相关信息的命令，它的常用方法如表 7-2 所示。

表 7-2　表描述类的常用方法

方法名称	返 回 值	描　述
addFamily(HColumnDescriptor family)	HTableDescriptor	增加列簇
getColumnFamilies()	HColumnDescriptor[]	获取列簇信息
getConfiguration()	Map<String, String>	获取配置信息列表
getTableName()	TableName	获取表名称
getValue(byte[] key)	String	获取元数据值
removeFamily(byte[] column)	HColumnDescriptor	删除列簇
setMaxFileSize(long maxFileSize)	HTableDescriptor	设置 Region 文件存储最大值，当大于此值时后，触发分裂
hasFamily(byte[] familyName)	boolean	是否包含该列簇

我们这里列出当前集群中的所有表，并且输出指定表的表结构信息。为方便学习，我们编写了一个主文件 HBaseBase，本次要完成的代码封装成一个该类的方法，并在该类的 main 方法中进行调用(后面所用的每一个演示都单独封装成这个类的方法，同样在 main 方法中进行调用)。

在 main 方法中设计了一个循环输入的方式，获取要添加的记录信息并提交。当输入 quit 时，自动退出。输入 list 时，则列出所有表。输入“show 表名”时，则列出指定表的结构信息。输入完成后，按 Enter 键提交。

示例代码如下所示：

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Map.Entry;
import java.util.Set;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.KeyValue;
```

```
import org.apache.hadoop.hbase.MasterNotRunningException;
import org.apache.hadoop.hbase.NamespaceDescriptor;
import org.apache.hadoop.hbase.ServerName;
import org.apache.hadoop.hbase.ClusterStatus;
import org.apache.hadoop.hbase.RegionLoad;
import org.apache.hadoop.hbase.ZooKeeperConnectionException;
import org.apache.hadoop.hbase.client.Delete;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.ServerLoad;
import org.apache.hadoop.hbase.client.HTablePool;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.filter.BinaryComparator;
import org.apache.hadoop.hbase.filter.CompareFilter;
import org.apache.hadoop.hbase.filter.QualifierFilter;
import org.apache.hadoop.hbase.filter.ValueFilter;
import org.apache.hadoop.hbase.util.Bytes;

import com.sun.jersey.core.reflection.MethodList.Filter;

//定义 HBase 的基础类库
public class HBaseBase {

    //定义基础配置信息
    private static Configuration hbaseconf = null;
    static {
        Configuration testconf = new Configuration();
        testconf.set("hbase.zookeeper.quorum", "iZ23d4by1laZ");
        testconf.set("hbase.zookeeper.property.clientPort", "2181");
        hbaseconf = HBaseConfiguration.create(testconf);
    }

    //此处，以下封装各类方法

    //查看所有表列表
    public void showtblist() throws MasterNotRunningException,
      ZooKeeperConnectionException, IOException
    {
        HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    1   String [] tblist = hbaseadmin.getTableNames();
        for(int i=0; i<tblist.length; i++)
            System.out.println("table [" + i + "] name" + tblist[i]);
    }
    //查看某张指定表的表结构
    public void showtbscheme(String tbname)
      throws MasterNotRunningException,
      ZooKeeperConnectionException, IOException
    {
        HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    2   if(hbaseadmin.tableExists(tbname))
        {
    3       HTableDescriptor tbdesc =
                hbaseadmin.getTableDescriptor(Bytes.toBytes(tbname));
            System.out.println(tbdesc);
        }
        else
        {
```

```
            System.out.println("table name " + tbname + " is not exist!");
        }
    }

    public static void main(String[] args) throws Exception {
        //此处进行方法调用
        HBaseBase hbasebase = new HBaseBase();
        //循环信息提交
4       BufferedReader reader =
          new BufferedReader(new InputStreamReader(System.in));
        while(true){
            System.out.print("myhbase:>");
            String strread = reader.readLine();
5           if(strread.equals("quit")) {
                System.out.println("Bye Bye!");
                return;
            }
6           String[] str = strread.split(",| ");
7           if(str[0].equals("show") && str.length==2) {
                //处理 show table 的格式请求
                hbasebase.showtbscheme(str[1]);
            }
8           else if(str[0].equals("list") && str.length==1) {
                //处理 show table 的格式请求
                hbasebase.showtblist();
            }
            else {
                System.out.println("Input Error!");
            }
        }
    }
}
```

代码解析与运行

代码 1 处：获取一个表名的数组集合，通过 for 循环输出各表的名称。

代码 2 处：用于判断是否存在该表，如果存在，返回 true，否则返回 false。

代码 3 处：获得指定表的 HTableDescriptor 类，该类中存储表结构描述信息。

代码 4 处：从键盘中获得用户输入。

代码 5 处：如果输入的是 quit，则直接退出。

代码 6 处：对输入进行切词，通过“,”或空格进行分隔。

代码 7 处：如果检测到输入的格式第一个字符串为 show，同时，整个切词后的数组长度为 2，则执行查看表结构的操作。

代码 8 处：如果检测到输入的格式第一个字符串为 list，同时，整个切词后的数组长度为 1，则列出当前所有表的操作。

示例代码的运行效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>list
table [0] name bigdata:bat_tb
table [1] name bigdata:emp_sign_tb
table [2] name bigdata:emp_tb
table [3] name exam_tb
table [4] name hbase_test
table [5] name tb2
```

```
table [6] name tb3
table [7] name testhbase
myhbase:>show hbase_test
'hbase_test', {NAME => 'f1', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER =>
'ROW', REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1',
MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'false',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}, {NAME =>
'f2', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE
=> '0', COMPRESSION => 'NONE', VERSIONS => '1', MIN_VERSIONS => '0', TTL =>
'FOREVER', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY
=> 'false', BLOCKCACHE => 'true'}
```

7.2.2 表的创建

HBaseAdmin 中的 createTable(HTableDescriptor desc)方法用于创建表，创建一张表需要涉及到两个信息类，一个是参数 HTableDescriptor desc，用于描述表的结构；另一个是 HColumnDescriptor，用于设定每一个列簇信息。

类 HColumnDescriptor 是列簇信息描述类，它用于描述列簇、属性等信息，支持属性值的设定、最大版本、加密等参数管理，它常用的方法如表 7-3 所示。

表 7-3　列簇描述类常用的方法

方法名称	返 回 值	描 述
getConfiguration()	Map<String, String>	获得配置信息列表
getEncryptionType()	String	获得加密算法名称
setEncryptionType(String algorithm)	HColumnDescriptor	设置列簇加密算法
removeConfiguration(String key)	void	删除某一配置信息
setMaxVersions(int maxVersions)	HColumnDescriptor	设置最大版本数
getNameAsString()	String	获取列簇名称
setTimeToLive(int timeToLive)	HColumnDescriptor	设置 TTL 值
setDurability(Durability d)	void	设置 WAL 级别

下面实现了一个方法 createhbasetable，用于从给定表名、列簇信息集合去创建指定的一张表。代码的主要流程是输入表名、表列簇集合，如果表已存在，则返回；如果表不存在，则构建表结构描述文件，并执行表的创建。

源代码如下所示：

```
//创建表，指定表名，表列簇信息
public void createhbasetable(String tbname, String[] family)
  throws Exception {
    HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    if (hbaseadmin.tableExists(tbname)) {
        System.out.println("table ["+tbname+"]Exists!");
    }
    else {
1       HTableDescriptor desc = new HTableDescriptor(tbname);
        for (int i=0; i<family.length; i++) {
2           desc.addFamily(new HColumnDescriptor(family[i]));
    }
        hbaseadmin.createTable(desc);
```

```
        System.out.println("create table ["+tbname+"]Success!");
    }
}
```

代码解析与运行

代码 1 处：初始化表描述对象，并以表名为传入参数。

代码 2 处：循环获得列簇名称，并通过 addFamily 方法添加至 desc 表描述对象中。

在 main 方法中，利用前面的循环输入方法，构建一个“create 表名，列簇 1，列簇 2, ..., 列簇 N”的命令集合，示例代码如下：

```
    ...
1   else if(str[0].equals("create") && str.length>=3) {
        //先格式化
2       String[] strfamily = new String[str.length-2];
        for(int i=2; i<str.length; i++)
3           strfamily[i-2] = str[i];
        //处理 create 请求
        hbasebase.createhbasetable(str[1], strfamily);
    }
    ...
```

代码 1 处：当用户输入为 create，以及输入的字符串数组大于 3 时，则执行本命令。

代码 2 处：创建列簇数组集合，长度为用户输入的字符串数组长度减去 2(减去 create 与表名)。

代码 3 处：对列簇数组集合进行赋值。

这里调用 create 命令，创建一张表名为 student，包括 school、work 两个列簇，创建完毕后，用 HBase Shell 命令进行验证，运行效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>create student,school,work
create table [student] Success!
[root@www hjava]# hbase shell
...
hbase(main):001:0> list 'student'
TABLE
student
1 row(s) in 1.0630 seconds
...
```

7.2.3　表的删除

HBaseAdmin 中的 deleteTable()方法可用于删除一组指定表，或删除符合某个正则表达式条件的一组表，删除表之前，需要用 disableTable()方法禁用该表。

下面实现了一个方法 deletehbasetable，用于删除给定的表名，代码的主要流程是输入表名，如果表已存在，则执行先禁用后删除的操作，如果不存在，则返回。

源代码如下所示：

```
//删除表，指定表名
public void deltehbasetable(String tbname) throws Exception {
    HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    if (hbaseadmin.tableExists(tbname)) {
        hbaseadmin.disableTable(tbname);
```

```
        hbaseadmin.deleteTable(tbname);
    }
    else {
        System.out.println("table ["+tbname+"] is not Exists!");
    }
}
```

代码解析与运行

在 main 方法中，利用前面的方法，构建一个“delete 表名”的命令集合，示例代码如下所示：

```
...
else if(str[0].equals("delete") && str.length==2) {
    //处理 delete table 请求
    hbasebase.deltehbasetable(str[1]);
}
...
```

删除之前创建的 student 表，同时利用 list 命令进行验证，运行效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>delete student
student is deleted!
myhbase:>list
table [0] name bigdata:bat_tb
table [1] name bigdata:emp_sign_tb
table [2] name bigdata:emp_tb
table [3] name exam_tb
table [4] name hbase_test
table [5] name tb2
table [6] name tb3
table [7] name testhbase
```

7.2.4 表的修改

对 HBase 表结构的修改，主要体现于列簇的增加或删除，通过 HBaseAdmin 中的 addColumn 方法，可以实现对列簇的增加，方法中的参数需要指定表名，以及添加的列簇结构描述信息，用 HColumnDescriptor 类进行描述；删除列簇通过 deleteColumn 方法，传入的参数包括表名与要列删除的列簇名。

下面设定了一个方法 alterhbasefamily(String tbname, String family, boolean flag)，通过 flag 进行控制，当为 true 时，执行增加列簇操作；为 false 时，执行删除列簇操作。

源代码如下所示：

```
//修改表，flag 为 true 时增加此列簇，为 false 时删除此列簇
public void alterhbasefamily(String tbname, String family, boolean flag)
  throws Exception {
    HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    if (hbaseadmin.tableExists(tbname))
    {
1       hbaseadmin.disableTable(tbname);
        HColumnDescriptor columndesc = new HColumnDescriptor(family);
        if(flag) {
            hbaseadmin.addColumn(tbname, columndesc);
            System.out.println("add family [" + family + "] Success!");
```

```
            } else {
                hbaseadmin.deleteColumn(tbname, family);
                System.out.println("delete family [" + family + "] Success!");
            }
2           hbaseadmin.enableTable(tbname);
        }
        else
        {
            System.out.println("table [" + tbname + "] is not Exists!");
        }
    }
```

代码解析与运行

代码 1 处：在修改表前，先禁用此表。

代码 2 处：修改完毕后，启动此表。

在 main 方法中调用该方法，实现对列簇的新增与删除命令，其中新增命令为“alteradd 表名, 列簇”；删除命令为“alterdel 表名, 列簇”，上述命令单次只能操作一个列簇。示例代码如下：

```
...
else if(str[0].equals("alteradd") && str.length==3) {
    //处理 alter add family 请求
    hbasebase.alterhbasefamily(str[1], str[2], true);
}
else if(str[0].equals("alterdel") && str.length==3) {
    //处理 alter delete family 请求
    hbasebase.alterhbasefamily(str[1], str[2], false);
}
...
```

实现对 student 表(注意在使用前先通过 create 命令创建)原有 work 列簇的删除，同时，新增 info 列簇，最后输出修改后的表结构，运行效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>alterdel student,work
delete family [work] Success!
myhbase:>alteradd student,info
add family [info] Success!
myhbase:>show student
'student', {NAME => 'info', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER =>
'ROW', REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1',
MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'false',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}, {NAME =>
'school', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1',
MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'false',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
```

小提示

关于修改表结构。

① 通过 HBaseAdmin 的 modifyTable(byte[] tableName, HTableDescriptor htd)方法，与 HTableDescriptor 中的 addFamily(HColumnDescriptor)、removeFamily(byte[] column)配合也能够实现表结构的修改。

② 在修改表结构前，要先禁用表，修改完毕后再启用表，以防止修改期间的数据修改，另外，在修改前，要确认数据是否需要保留。

7.2.5 命名空间

HBaseAdmin 中，支持命名空间的编程管理，可以实现对命名空间的信息查看、创建、删除、描述信息修改等操作。

1. 命名空间的查看

查看命名空间的方法为 listNamespaceDescriptors()，调用该方法返回当前所有命名空间(NamespaceDescriptor)的列表。

类 NamespaceDescriptor 是命名空间描述类，它包括命名空间名称、命名空间表、命名空间各类元数据，以及管理上述信息的方法。它常用的方法如表 7-4 所示。

表 7-4 命名空间描述类的常用方法

方法名称	返 回 值	描 述
create(String name)	NamespaceDescriptor.Builder	创建命名空间
setConfiguration(String key, String value)	void	设置命名空间属性键与值
removeConfiguration(String key)	void	删除某一配置信息
getConfiguration()	Map<String, String>	获得配置信息列表
getConfigurationValue(String key)	String	获取指定配置键的值
getName()	String	获取命名空间名称

这里实现一个查看当前集群所有命名空间，以及每个命名空间描述信息的示例。源代码如下所示：

```
//查看所有命名空间
public void listhbasenamespace() throws Exception {
    HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    NamespaceDescriptor[] nslist = hbaseadmin.listNamespaceDescriptors();
    for(int i=0; i<nslist.length; i++) {
        System.out.println(
          "NameSpace ID [" + i + "] " + " name: " + nslist[i].getName());
1       System.out.println(
          "descriptor : " + nslist[i].getConfiguration().toString());
    }
}
```

代码 1 处用于查看指定命名空间的配置描述信息。在 main 方法中创建 list_namespace 命令调用该方法，调用代码如下所示：

```
...
else if(str[0].equals("list_namespace") && str.length==1) {
    //处理 list namespace 的格式请求
    hbasebase.listhbasenamespace();
}
...
```

运行效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>list_namespace
NameSpace ID [0]  name: bigdata
descriptor: {creattime=2016-01-30}
NameSpace ID [1]  name: default
descriptor: {}
NameSpace ID [2]  name: hbase
descriptor: {}
```

2. 命名空间的创建

创建命名空间的方法是 createNamespace(NamespaceDescriptor descriptor)，这里的 NamespaceDescriptor 用于定义命名空间，可包括的信息有命名空间名称、命名空间描述。

下面实现一个方法，用于创建指定名称的命名空间，以及添加一条描述信息。在创建命名空间之前，应该先要判断命名空间是否已存在，因此，这里先实现一个查找指定的命名空间是否存在的方法 findhbasenamespace(String nsname)，该方法基于 listhbasenamespace()方法改写。

源代码如下所示：

```
// 查找某一个命名空间是否存在
public boolean findhbasenamespace(String nsname) throws Exception
{
   HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);

   NamespaceDescriptor[] nslist = hbaseadmin.listNamespaceDescriptors();
   for(int i=0; i<nslist.length; i++) {
1      if(nsname.equals(nslist[i].getName()))
          return true;
   }
   return false;
}

//创建指定的命名空间
public void createhbasenamespace(String nsname, String key,
  String keyvalue) throws Exception {
   HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
   if(this.findhbasenamespace(nsname)) {
      System.out.println("namespace " + nsname + " is exists!");
   }
   else {
2      NamespaceDescriptor ns =
        NamespaceDescriptor.create(nsname).build();
3      ns.setConfiguration(key, keyvalue);
      hbaseadmin.createNamespace(ns);
      System.out.println("create namespace " + nsname + " ok!");
   }
}
```

代码 1 处：用于匹配指定的命名空间是否存在。

代码 2 处：用于构造命名空间。

代码 3 处：用于填充命名空间描述信息。

在 main 方法中创建 create_namespace 命名空间名称、属性、属性值的命令，并调用创建命名空间方法。

相关的代码如下所示：

```
...
else if(str[0].equals("create_namespace") && str.length==4) {
    //处理 create namespace 的格式请求
    hbasebase.createhbasenamespace(str[1], str[2], str[3]);
}
...
```

这里创建一个名为ns，属性信息为<"desc": "create_namespace">的命名空间，并用HBase Shell命令进行验证。运行效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>create_namespace ns,desc,create_namespace
create namespace ns ok!
...
hbase(main):003:0> describe_namespace 'ns'
DESCRIPTION
{NAME => 'ns', desc => 'create_namespace'}
1 row(s) in 0.0230 seconds
```

3. 命名空间的修改

利用modifyNamespace(NamespaceDescriptor descriptor)可以修改命名空间的属性信息，通过descriptor设定要覆盖的属性信息，即可以完成修改操作。

下面实现一个命名空间修改的方法，输入参数包括命名空间名称、要覆盖的属性键与值，源代码如下所示：

```
//修改指定的命名空间
public void modifyhbasenamespace(String nsname, String key, String keyvalue)
  throws Exception {
    HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    if(this.findhbasenamespace(nsname)) {
        NamespaceDescriptor ns =
          NamespaceDescriptor.create(nsname).build();
        ns.setConfiguration(key, keyvalue);
        hbaseadmin.modifyNamespace(ns);
        System.out.println("modify namespace " + nsname + " ok!");
    }
    else {
        System.out.println("namespace " + nsname + " is not exists!");
    }
}
```

在main方法中创建一个“modify_namespace 命名空间, 属性, 属性值”的命令，调用该方法。相关的代码如下所示：

```
...
else if(str[0].equals("modify_namespace") && str.length==4) {
    //处理 modify namespace 的格式请求
    hbasebase.modifyhbasenamespace(str[1], str[2], str[3]);
}
...
```

修改ns命名空间属性信息，将“desc属性”改成“create 创建时间”，并设定值为20160123，并借助HBase Shell命令进行验证。运行效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>modify_namespace ns,create,20160123
modify namespace ns ok!
```

```
...
hbase(main):002:0> describe_namespace 'ns'
DESCRIPTION
{NAME => 'ns', create => '20160123'}
1 row(s) in 0.0120 seconds
```

4. 命名空间的删除

删除命名空间的方法是 deleteNamespace(String name)，这里实现一个删除指定命名空间的方法，源代码如下所示：

```
//删除指定的命名空间
public void deletehbasenamespace(String nsname) throws Exception {
    HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    if(this.findhbasenamespace(nsname)) {
1       hbaseadmin.deleteNamespace(nsname);
        System.out.println("delete namespace " + nsname + " ok!");
    }
    else {
        System.out.println("namespace " + nsname + " is not exists!");
    }
}
```

代码 1 处：执行删除命名空间的操作。

在 main 方法中，创建“delete_namespace 命名空间”的命令，并调用该方法。相关代码如下所示：

```
...
else if(str[0].equals("delete_namespace") && str.length==2) {
    //处理 delete namespace 的格式请求
    hbasebase.deletehbasenamespace(str[1]);
}
...
```

这里删除 ns 命名空间，并借助前面编写的 list_namespace 命令进行验证。运行效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>delete_namespace student
delete namespace ns ok!
myhbase:>list_namespace
NameSpace ID [0]  name: bigdata
descriptor: {creattime=2016-01-30}
NameSpace ID [1]  name: default
descriptor: {}
NameSpace ID [2]  name: hbase
descriptor: {}
```

7.3 数 据 编 程

HTable 类是 HBase 中重要的表数据操作类，拥有对数据增加、删除、修改、查询等的接口，使用 HTable 类前需要先初始化，所需要的参数是 HBase 集群连接配置文件，以及数据表名。

相关的代码如下所示：

```
HTable htb = new HTable(hbaseconf, tbname);  //tbname 是表名称
```

类 HTable 的常用方法如表 7-5 所示，本小节将围绕数据的增加、删除、修改、查询、过滤等操作进行实例讲解。

表 7-5　HTable 的常用方法

方法名称	返 回 值	描　述
append(Append append)	Result	追加数据
batch(List<? extends Row> actions, Object[] results)	void	批量执行增加、追加、删除、修改等操作
delete(Delete delete)	void	删除某一列或行的值
exists(Get get)	boolean	判断某列是否存在
get(Get get)	Result	获得一行或一行某列簇、某列的数据
getConfiguration()	Configuration	获得表配置对象
getName()	TableName	获得表名称
getScanner(byte[] family)	ResultScanner	获得指定列簇的结果集
getScanner(Scan scan)	ResultScanner	通过 scan 对象获取全表结果集
getTableDescriptor()	HTableDescriptor	获得表描述信息
increment(Increment increment)	Result	增加某行中某列或列簇值
put(Put put)	void	增加数据
setWriteBufferSize(long writeBufferSize)	void	设置写缓存的大小
close()	void	关闭表连接

7.3.1　数据的增加

调用 HTable 的 put(Put put)方法，来提交数据，传入参数 Put 类是需要实例化数据信息，通常包括行键值、列簇、字段名、字段值等信息，当然，也可以加入时间戳，如果不加时间戳，则系统会自动引入。类 Put 用于格式化要提交的数据，它的常用方法如表 7-6 所示。

表 7-6　Put 类常用的方法

方法名称	返 回 值	描　述
add(Cell kv)	Put	增加一个列值，Cell 是一个列值对象
addColumn(byte[] family, byte[] qualifier, byte[] value)	Put	向指定列簇中的某列增加值
get(byte[] family, byte[] qualifier)	List<Cell>	获得指定列簇，指定列的值集合
has(byte[] family, byte[] qualifier)	boolean	判断列簇中是否包括某列
setACL(Map<String, Permission> perms)	Put	设定权限
setAttribute(String name, byte[] value)	Put	设置属性
setTTL(long ttl)	Put	设置 TTL 值

这里实现一个 insertsinglerecord 方法，能够实现一列记录的添加，源代码如下所示：

```
// 插入一列记录
public  void insertsinglerecord(String tbname, String rowKey,
  String familyname, String columnname, String value) throws Exception {
    try {
        HTable htb = new HTable(hbaseconf, tbname);
1       Put put = new Put(Bytes.toBytes(rowKey));
2       put.add(Bytes.toBytes(familyname),
          Bytes.toBytes(columnname),Bytes.toBytes(value));
        htb.put(put);
        System.out.println("Status: [Sucess] insert recored "
          + rowKey + " to table " + tbname + " ok...");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

代码解析与运行

在 main 方法中创建一个“put 表名, 行键值, 列簇名, 字段名”的命令，并调用数据增加方法。源代码如下所示：

```
...
else if(str[0].equals("put") && str.length==6) {
    //处理 put 请求
    hbasebase.insertsinglerecord(str[1], str[2], str[3], str[4], str[5]);
}
...
```

测试并运行，这里提交五列记录，同时，借助 HBase Shell 查看表记录进行验证，效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>put student,1,info,age,20
Status: [Sucess] insert recored 1 to table student ok...
myhbase:>put student,1,info,name,tom
Status: [Sucess] insert recored 1 to table student ok...
myhbase:>put student,2,info,,student information
Status: [Sucess] insert recored 2 to table student ok...
myhbase:>put student,2,info,age,23
Status: [Sucess] insert recored 2 to table student ok...
myhbase:>put student,2,info,name,tom
Status: [Sucess] insert recored 2 to table student ok...
...
hbase(main):008:0> scan 'student'
ROW                          COLUMN+CELL
 1         column=info:age, timestamp=1459132877213, value=20
 1         column=info:name, timestamp=1459132885102, value=tom
 2         column=info:, timestamp=1459132909940, value=student information
 2         column=info:age, timestamp=1459133055925, value=23
 2         column=info:name, timestamp=1459133106700, value=tom
2 row(s) in 0.0300 seconds
```

利用本方法，除数据增加之外，还可以完成数据的修改，例如，当前行键值 2 的 info:name 为 tom，要改成 jim，利用本方法即可完成操作，如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>put student,2,info,name,jim
```

```
Status: [Sucess] insert record 2 to table student ok....
hbase(main):016:0> get 'student','2'
COLUMN                         CELL
 info:               timestamp=1459132909940, value=student information
 info:age            timestamp=1459133055925, value=23
 info:name           timestamp=1459135434239, value=jim
3 row(s) in 0.0220 seconds
```

小总结

关于行锁。

① 大规模业务并发请求时，可借助显示行锁 RowLock 进行控制，以确保数据的一致性。

② 在数据操作前调用 lockRow，在完成操作后调用 unlockRow。

③ hbase-site.xml 中的参数 hbase.regionserver.lease.period 用于控制行锁默认锁定时间，默认行锁的时间为 1 分钟。

7.3.2 单行查询

调用 HTable 的 get(Get get)方法，可以查询某一行所有列簇的数据，传入的参数 Get 需要指定行键值。类 Get 用于格式要查询的元数据，通过此对象设定查询条件。常用方法如表 7-7 所示。

表 7-7 Get 类的常用方法

方法名称	返 回 值	描 述
addFamily(byte[] family)	Get	设定列簇的名称
addColumn(byte[] family, byte[] qualifier)	Get	设定列簇以及列名称
getMaxVersions()	int	获得最大版本数
getRow()	byte[]	获得某一行数据
setACL(Map<String, Permission> perms)	Get	设定权限
setColumnFamilyTimeRange(byte[] cf, long minStamp, long maxStamp)	Get	设定查询的时间戳区间
setFilter(Filter filter)	Get	设置过滤器

这里实现一个 getsinglerecord 方法，能够实现通过输入表名、行键值，查询获得结果记录，源代码如下所示：

```
//查询一行记录
public void getsinglerecord(String tbname, String rowkey)
  throws IOException {
    HTable htb = new HTable(hbaseconf, Bytes.toBytes(tbname));
    Get get = new Get(Bytes.toBytes(rowkey));
1   Result res = htb.get(get);
    System.out.println("family\tcolumn\tvalue\tversion\ttimstamp ");
2   for (KeyValue kv : res.list()) {
        System.out.print(Bytes.toString(kv.getFamily()));
        System.out.print("\t" + Bytes.toString(kv.getQualifier()));
        System.out.print("\t" + Bytes.toString(kv.getValue()));
        System.out.print("\t" + kv.getMvccVersion());
```

```
        System.out.println("\t" + kv.getTimestamp());
    }
}
```

代码解析与运行

代码 1 处：获得查询后的结果集合。

代码 2 处：循环提取每个列数据。

在 main 方法中，创建“get 表名, 行键值”的命令，并调用前面的单行查询方法。源代码如下所示：

```
...
else if(str[0].equals("get") && str.length==3)
{
    //处理 get 请求
    hbasebase.getsinglerecord(str[1], str[2]);
}
else {
    ...
}
...
```

这里查询前面提交的数据，效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>get student,1
family  column  value  version  timstamp
info  age  20  0  1459132877213
info  name  tom  0  1459132885102
myhbase:>get student,2
family  column  value  version  timstamp
info   student information  0  1459132909940
info  age  23  0  1459133055925
info  name  jim  0  1459135434239
```

7.3.3　集合查询

调用 HTable 的 getScanner(Scan scan)方法，可以查询表集合的数据，需要传入的参数为 Scan 类，在 Scan 类里面，可以指定全部的表数据，也可以指定某行键值区间的数据。

类 Scan 用于格式化要查询数据集合的数据，通过此对象设定集合查询条件。它常用的方法如表 7-8 所示。

表 7-8　Scan 类的常用方法

方法名称	返 回 值	描　述
addFamily(byte[] family)	Scan	设定列簇的名称
addColumn(byte[] family, byte[] qualifier)	Scan	设定列簇以及列的名称
getBatch()	int	获取数据集合的条数
getFamilies()	byte[][]	获取查询到的所有列簇名称
getMaxVersions()	int	获得最大版本数
setStartRow(byte[] startRow)	Scan	设定查询起始行

续表

方法名称	返 回 值	描 述
setStopRow(byte[] startRow)	Scan	设定查询结束行
setACL(Map<String, Permission> perms)	Scan	设定权限
setTimeRange(long minStamp, long maxStamp)	Scan	设定查询时间戳区间
setBatch(int batch)	Scan	设置 next()返回的记录条数
setFilter(Filter filter)	Scan	设置过滤器
setRowPrefixFilter(byte[] rowPrefix)	Scan	设置行过滤器

这里实现一个 getresultset 方法，能够实现从输入表名、行键值区间获得结果集合记录，源代码如下所示：

```
//集合查询
public void getresultset(String tbname, String control, String startrowkey,
  String endrowkey) throws IOException {
    HTable htb = new HTable(hbaseconf, Bytes.toBytes(tbname));
    Scan scan = new Scan();
    if(control.equals("region")) {
1       scan.setStartRow(Bytes.toBytes(startrowkey));
2       scan.setStopRow(Bytes.toBytes(endrowkey));
    }
    ResultScanner res = null;
    try {
        res = htb.getScanner(scan);
        for (Result r : res) {
            for (KeyValue kv : r.list()) {
                System.out.println("row:" + Bytes.toString(kv.getRow()));
                System.out.println(
                  "family:" + Bytes.toString(kv.getFamily()));
                System.out.println(
                  "qualifier:" + Bytes.toString(kv.getQualifier()));
                System.out.println(
                  "value:" + Bytes.toString(kv.getValue()));
                System.out.println("timestamp:" + kv.getTimestamp());
                System.out.println("---------------------------------");
            }
        }
    } finally {
        res.close();
    }
}
```

代码解析与运行

代码 1 处：用于设定查询区间的初始行键值。

代码 2 处：用于设定查询区间的结束行键值。

在 main 方法中设定的输入格式为“scan 表名”或“scan 表名，初始行键值，结束行键值”的命令，并调用查询方法。

源代码如下所示：

```
...
else if(str[0].equals("scan") && str.length==2)
{
```

```
        //处理 scan 请求
        hbasebase.getresultset(str[1], "", "", "");
    }
    else if(str[0].equals("scan") && str.length==4)
    {
        //处理 scan 请求
        hbasebase.getresultset(str[1], "region", str[2], str[3]);
    }
    ...
```

测试并运行，这里查询前面提交的数据，效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>scan student
row:1    info:age value:20 timestamp:1459132877213
row:1    info:name value:tom timestamp:1459132885102
row:2    info: value:student information timestamp:1459132909940
row:2    info:age value:23 timestamp:1459133055925
row:2    info:name value:jim timestamp:1459135434239
myhbase:>scan student,1,1
row:1    info:age value:20 timestamp:1459132877213
row:1    info:name value:tom timestamp:1459132885102
```

7.3.4　过滤器

前面提到的单行查询、集合查询是按照行键值或者列簇、字段进行数据显示，设想，在大规模的 HBase 集群数据中去执行这样的查询，将会极为低效，因为返回的结果集合会是比较庞大的，不利于数据处理与分析。

在 HBase 中，没有提供像 SQL 查询语句那样方便的接口，但为方便数据精细化操作，提供了过滤器这样的接口，即对于查询结果，按指定的方法进行过滤后返回结果集，在 Get、Scan 中均支持过滤器。

过滤器的调用需要实例化 Filter 类，通过不同的过滤器进行实例化以达到不同的过滤目的，实例化后，在 Get 或 Scan 中通过 setFilter 方法加载过滤器，过滤器的基类是 Filter。在 HBase 中支持多种过滤器机制，比较常见的是比较过滤器 CompareFilter 类，在该类中定义了常用的比较运算符，即用设定的值进行比较，满足条件的记录被返回，所有比较运算符如表 7-9 所示。

表 7-9　比较运算符

名　称	描　述
CompareFilter.CompareOp.EQUAL	等于
CompareFilter.CompareOp.GREATER	大于
CompareFilter.CompareOp.GREATER_OR_EQUAL	大于等于
CompareFilter.CompareOp.LESS	小于
CompareFilter.CompareOp.LESS_OR_EQUAL	小于或等于
CompareFilter.CompareOp.NO_OP	排除一切值
CompareFilter.CompareOp.NOT_EQUAL	不等于

除上面的比较运算符外，还定义了比较器，常见的比较器包括：BinaryComparator(字节

比较器)、BinaryPrefixComparator(字节前缀比较器)、RegexStringComparator(正则表达式比较器)、NullComparator(空值比较器)、BitComparator(比特位比较器)、SubstringComparator(子串比较器)等。

这里以字段名称过滤器(QualifierFilter)为例，设计一个方法，返回指定字段名的数据集。相关源代码如下所示：

```
//字段名称比较过滤器查询
public void getfilterresult(String tbname, String column)
  throws IOException {
    HTable htb = new HTable(hbaseconf, Bytes.toBytes(tbname));
    Scan scan = new Scan();
1   QualifierFilter fr = new QualifierFilter(
      CompareFilter.CompareOp.EQUAL,
      new BinaryComparator(Bytes.toBytes(column)));
2   scan.setFilter((org.apache.hadoop.hbase.filter.Filter)fr);
    ResultScanner res = null;
    try {
        res = htb.getScanner(scan);
        for (Result r : res) {
            for (KeyValue kv : r.list()) {
                System.out.print("row:" + Bytes.toString(kv.getRow()));
                System.out.print("\t" + Bytes.toString(kv.getFamily()));
                System.out.print(":"  + Bytes.toString(kv.getQualifier()));
                System.out.println(
                  " value:" + Bytes.toString(kv.getValue()));
            }
        }
    } finally {
        res.close();
    }
}
```

代码解析与运行

代码 1 处：用于设定 QualifierFilter 过滤器，该过滤器的作用是输出所有字段名为变量 column 的数据。

代码 2 处：设定过滤器。

在 main 方法中设定的输入格式为“quafilter 表名，字段名”并调用过滤器查询方法。源代码如下所示：

```
...
else if(str[0].equals("quafilter") && str.length==3)
{
    //处理过滤器请求
    hbasebase.getfilterresult(str[1], str[2]);
}
...
```

测试并运行，效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>quafilter student,age
row:1  info:age value:20 timestamp:1459132877213
row:2  info:age value:23
```

7.3.5　数据删除

调用 HTable 的 delete(Delete delete)方法，可以删除表集数据，需要传入的参数为 Delete 类，在 Delete 类里面，设置可以删除的列对象。

类 Delete 用于格式化要删除的列对象，通过此对象设定删除对象的条件。它的常用方法如表 7-10 所示。

表 7-10　Delete 类常用的方法

方法名称	返 回 值	描　述
addFamily(byte[] family)	Delete	设定要删除的列簇名称，默认删除全部版本
addColumn(byte[] family, byte[] qualifier)	Delete	设定要删除的列簇以及列名称，默认删除最近版本
setTimestamp(long timestamp)	Delete	设定时间戳
setACL(Map<String,Permission> perms)	Scan	设定权限

这里实现一个 delete 方法，能够实现通过输入表名、行键值区间获得结果集合记录，源代码如下所示：

```
//删除一列或一行数据，如果 control 为 true，则删除一列，否则删除一行
public void deletehbasecolumn(String tbname, boolean control, String rowkey,
  String family, String column) throws IOException {
    HTable htb = new HTable(hbaseconf, Bytes.toBytes(tbname));
1   Delete deleteColumn = new Delete(Bytes.toBytes(rowkey));
    if(control) {
2       deleteColumn.deleteColumns(
          Bytes.toBytes(family), Bytes.toBytes(column));
    }
    htb.delete(deleteColumn);
    if(control) {
        System.out.println(
          "delete column " + rowkey + " " + family + ":" + column + " OK!");
    } else {
        System.out.println("delete row " + rowkey + " OK!");
    }
}
```

代码解析与运行

代码 1 处：用于构造要删除的类。

代码 2 处：用于设定要删除的列字段。

在 main 方法中创建“delete 表名, 行键值”或“delete 表名, 行键值, 列簇, 字段”的命令并调用删除方法，用于删除一行或一列记录。

源代码如下所示：

```
...
else if(str[0].equals("delete") && str.length==3)
{
    //处理 delete 请求
    hbasebase.deletehbasecolumn(str[1], false, str[2], "", "");
}
```

```
else if(str[0].equals("delete") && str.length==5)
{
    //处理 delete 请求
    hbasebase.deletehbasecolumn(str[1], true, str[2], str[3], str[4]);
}
...
```

测试并运行，这里查询前面提交的数据，效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>delete student,1,info,age
delete row 1 column:  info:age  OK!
myhbase:>scan student
row:1  info:name value:tom timestamp:1459132885102
row:2  info: value:student information timestamp:1459132909940
row:2  info:age value:23 timestamp:1459133055925
row:2  info:name value:jim timestamp:1459135434239
myhbase:>delete student,1
delete row 1 OK!
myhbase:>scan student
row:2  info: value:student information timestamp:1459132909940
row:2  info:age value:23 timestamp:1459133055925
row:2  info:name value:jim timestamp:1459135434239
```

7.4　集群与优化编程

面向大规模的数据存取与管理时，涉及到对 HBase 集群的编程开发，同样也涉及到通过多表或表池技术来提升数据访问效率，本节将重点介绍这些内容。

7.4.1　集群管理

利用 HBaseAdmin 类，可以实现对集群的管理，这种管理包括对状态的监测、服务关闭、Region 区域的调整、快照管理、数据迁移等，这里以服务、Region 为示例，介绍集群的管理操作。

1. 服务管理

调用 HBaseAdmin 类中如表 7-11 所示的方法，可以关闭 Region 服务器，可以关闭 Master，甚至关闭整个集群。

表 7-11　服务管理方法

方法名称	返 回 值	描　述
stopMaster()	void	停止 Master 服务
stopRegionServer(String hostnamePort)	void	停止指定名称的 Region 服务器
shutdown()	void	停止集群服务
isMasterRunning()	boolean	Master 服务是否运行

这里设计一个示例，输入 stop 命令，stop all 为关闭整个集群，stop master 为关闭 Master 服务器，“stop Region 服务器名称”为关闭指定的 Region 服务器。

源代码如下所示：

```
//关闭集群、关闭 Master，关闭 RegoinServer
public void stopservice(String name) throws Exception {
    HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    if (name.equals("all")) {
        hbaseadmin.shutdown();
        System.out.println("shut down cluster!");
    }
    else if(name.equals("master")) {
        hbaseadmin.stopMaster();
        System.out.println("shut down master!");
    }
    else {
        hbaseadmin.stopRegionServer(name);
        System.out.println("shut down regionserver " + name + " !");
    }
}
```

main 方法中的相关代码如下所示：

```
...
else if(str[0].equals("stop") && str.length==2)
{
    //处理 stop all 或 stop hostname 请求
    hbasebase.stopservice(str[1]);
}
...
```

运行效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:>stop all
shut down cluster!
...
```

2. Region 管理

针对 Region 的管理，在 HBaseAdmin 类中提供了比较丰富的接口，包括 Region 的关闭、分裂、移动、合并等，通过这些方法，能够方便用户在运维 HBase 集群时编写自动化调度与管理任务。

比较典型的 Region 操作方法如表 7-12 所示。

表 7-12　Region 的管理方法

方法名称	返 回 值	描　述
closeRegion(byte[] regionname, String serverName)	void	关闭指定服务器上的 Region
compactRegion(byte[] regionName)	void	压缩一个独立的 Region
getTableRegions(TableName tableName)	List<HRegionInfo>	获取表中的所有 Region 清单
offline(byte[] regionName)	void	将指定的 Region 下线
mergeRegions(byte[] nameOfRegionA, byte[] nameOfRegionB, boolean forcible)	void	合并两个 Region
splitRegion(byte[] regionName)	void	拆分 Region

续表

方法名称	返 回 值	描 述
assign(byte[] regionName)	void	分配一个 Region
move(byte[] encodedRegionName, byte[] destServerName)	void	移动 Region 到目标服务器

这里设计一个示例，当输入 split 的命令，语法为“split region 名称”，则实现指定的 Region 的拆分。

源代码如下所示：

```
//split region 管理
public void splitregion(String regionname) throws Exception {
    HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    hbaseadmin.split(regionname);
    System.out.println("Split " + regionname + " OK !");
}
```

main 方法中的相关代码如下所示：

```
...
else if(str[0].equals("split") && str.length==2) {
    //处理 split 请求
    hbasebase.splitregion(str[1]);
}
...
```

运行效果如下所示：

```
[root@www hjava]# java HBaseBase
myhbase:> split c663d751446746badc7653c684670c98
Split c663d751446746badc7653c684670c98 OK !
```

7.4.2 集群监测

HBase 中，利用 HBaseAdmin 类进行状态监测主要包括以下两个方面：

一是集群状态层面，涉及到集群状态、RegionServer 服务器数量、可用 RegionServer 数量、不可用 RegionServer 数量、Region 数量等，监测集群要使用 ClusterStatus 类。

二是 RegionServer 状态层面，涉及到服务器名称、开放端口、服务器的 Region 数量、每个 Region 的存储及资源占用情况，这主要涉及到三个类，分别是 ServerName、ServerLoad、RegionLoad 类。

监测是调度 HBase 集群的基础，通过监测，可以通过获取不同粒度的信息以精细定位存在的问题，下面实现两个方法，作用分别是：实现集群状态以及服务器、Region 数量统计；实现当前所有服务器或某一个服务器的状态以及服务器上的 Region 状态信息。

(1) 集群状态监测使用前，先导入 org.apache.hadoop.hbase.ClusterStatus 类，状态查看源代码如下所示：

```
//集群状态
public void clusterstatus() throws Exception
{
    hbaseadmin = new HBaseAdmin(hbaseconf);
```

```
        ClusterStatus cs = hbaseadmin.getClusterStatus();
        System.out.println("————————集群状态—————————————");
        System.out.println("HBase 版本:" + cs.getHBaseVersion());
        System.out.println("软件版本:" + cs.getVersion());
        System.out.println("集群编号:" + cs.getClusterId());
        System.out.println("集群 Master 列表:" + cs.getMaster());
        System.out.println("集群 backup Master 数量:"
          + cs.getBackupMastersSize());
        System.out.println("集群 backup Master 列表:" + cs.getBackupMasters());
        System.out.println("集群平均负载:" + cs.getAverageLoad());
        System.out.println("存活服务器数量:" + cs.getServersSize());
        System.out.println("存活服务器列表:" + cs.getServers());
        System.out.println("宕机数量:" + cs.getDeadServers());
        System.out.println("宕机列表:" + cs.getDeadServerNames());
        System.out.println("Region 数量:" + cs.getRegionsCount());
        System.out.println("业务请求数量:" + cs.getRequestsCount());
        System.out.println("Master 协处理器:" + cs.getMasterCoprocessors());
        System.out.println("—————————————————————————————");
    }
```

(2) 服务器状态在使用前，需要导入 org.apache.hadoop.hbase.ServerName、org.apache.hadoop.hbase.ServerLoad、org.apache.hadoop.hbase.RegionLoad 类，方法的参数只有一个，即服务器名称，当该变量为 all 时，则输出所有服务器状态信息。

源代码如下所示：

```
//服务器状态
public void serverstatus(String servername)
  throws MasterNotRunningException, ZooKeeperConnectionException,
  IOException {
    HBaseAdmin hbaseadmin = new HBaseAdmin(hbaseconf);
    ClusterStatus cs = hbaseadmin.getClusterStatus();
    int i = 1;
    //循环获取每个 RegionServer
    for(ServerName sn : cs.getServers())
    {
        //如果当前传递的是 all 或者传递的是当前服务器列表，则输出
        if(servername.equals("all")
          || sn.getHostname().equals(servername))
        {
            System.out.println("服务器编号: " + i++);
            System.out.println("服务器信息:" + sn.getServerName());
            System.out.println("起始码: " + sn.getStartcode());

            //显示服务器负载情况
            ServerLoad sl = cs.getLoad(sn);
            System.out.println("请求数量:" + sl.getNumberOfRequests());
            System.out.println("读请求数:" + sl.getReadRequestsCount());
            System.out.println("每秒请求:" + sl.getRequestsPerSecond());
            System.out.println("写请求数: " + sl.getWriteRequestsCount());
            System.out.println("负载情况:" + sl.getLoad());
            System.out.println("Max Heap(MB):" + sl.getMaxHeapMB());
            System.out.println("Mem Store(MB):" + sl.getMemstoreSizeInMB());
            System.out.println("Region 数量: " + sl.getNumberOfRegions());
            System.out.println("根索引大小(KB):" + sl.getRootIndexSizeKB());
            System.out.println("存储文件数量" + sl.getStorefiles());
            System.out.println(
              "存储文件大小(MB):" + sl.getStorefileIndexSizeInMB());

            //循环获取每个 Region 并显示
            int j = 1;
```

```
            for(Entry<byte[], RegionLoad> ey :
              sl.getRegionsLoad().entrySet())
            {
                RegionLoad rl = ey.getValue();
                System.out.println("Region 编号: " + j++);
                System.out.println("Region 名称:" + rl.getNameAsString());
                System.out.println("总请求数:" + rl.getRequestsCount());
                System.out.println("读请求数:" + rl.getReadRequestsCount());
                System.out.println("写请求数:" + rl.getWriteRequestsCount());
                System.out.println(
                  "Mem Store(MB):" + rl.getMemStoreSizeMB());
                System.out.println(
                  "Store File(MB):" + rl.getStorefileSizeMB());
                System.out.println("Store Files 数量:" + rl.getStorefiles());
                System.out.println(
                  "Root Index(KB):" + rl.getRootIndexSizeKB());
                System.out.println("-----------------------------------");
            }
        }
    }
}
```

在 main 方法中，设计了 clusterstatus、server 命令，分别用于显示集群与服务器的状态，针对 server 命令，如果输入的服务器名称为 all，则显示集群中所有服务器信息，如果要显示某一个服务器，则后跟服务器名称即可。

main 方法中，源代码如下所示：

```
...
else if(str[0].equals("clusterstatus") && str.length==1)
{
    //处理 clusterstauts 请求
    hbasebase.clusterstatus();
}
else if(str[0].equals("server") && str.length==2)
{
    //处理 server all 或 server hostname 请求
    hbasebase.serverstatus(str[1]);
}
...
```

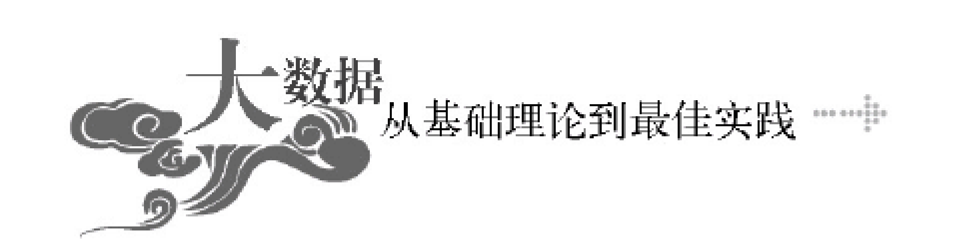
运行效果如下所示(部分显示有删减)：

```
[root@www hjava]# java HBaseBase
myhbase:>clusterstatus
———————————————集群状态————————————————
HBase 版本:0.98.3-hadoop1
软件版本:2
集群编号:1e84fe97-f16a-4c2c-ad09-dc01f26f2609
集群 Master 列表:iZ23d4by1laZ,60000,1457240251932
集群 backup Master 数量:0
集群 backup Master 列表:[]
集群平均负载:12.0
存活服务器数量:1
存活服务器列表:[iZ23d4by1laZ,60020,1457240254015]
宕机数量:0
宕机列表:[]
Region 数量:12
业务请求数量:25936
Master 协处理器:[Ljava.lang.String;@6d5a1741
---------------------------------------------------------------------
```

```
myhbase:>server all
服务器编号: 1
服务器名称:iZ23d4by1laZ,60020,1457240254015
起始码: 1457240254015
负载情况:12
Max Heap(MB):983
Mem Store(MB):0
Region 数量: 12
请求数量:0
读请求数:87127
每秒请求:0.0
写请求数: 138
根索引大小(KB):0
存储文件数量13
存储文件大小(MB):0
Region 编号: 1
Region 名称:hbase:meta,,1
Mem Store(MB):0
Store File(MB):0
Store Files 数量:1
Root Index(KB):0
总请求数:86672
读请求数:86636
写请求数:36
----------------------------------------
Region 编号: 2
Region 名称:hbase:namespace,,
1451358713278.2dad5136e3f76dab12bccd76c213121b.
Mem Store(MB):0
Store File(MB):0
Store Files 数量:2
Root Index(KB):0
总请求数:182
读请求数:164
写请求数:18
----------------------------------------
Region 编号: 3
Region 名称:student,,1458986225866.1a124f0d371354a8c34bab9f89cd9cf6.
Mem Store(MB):0
Store File(MB):0
Store Files 数量:2
Root Index(KB):0
总请求数:69
读请求数:52
写请求数:17
----------------------------------------
...
Region 编号: 12
Region 名称:testhbase,,1453080469131.0c3da9d154163bbb2b106ae878acc9cc.
Mem Store(MB):0
Store File(MB):0
Store Files 数量:2
Root Index(KB):0
...
```

7.4.3　多表与表池

HTable 是 HBase 最常用的对象，针对数据的操作要借助该对象。创建一个 HTable 对象比较耗时，在实际的 HBase 集群应用开发中，每秒要承担大量 I/O 请求，很难为每一次操



作独立创建一个 HTable 对象，那极不现实。

在实际读写操作时，有两种途径可进行性能优化，一种是批量创建多个表对象，以多线程的形式使用，另一种是使用表池技术，下面分别进行介绍。

1. 多表技术

同时创建多个表对象，并设定多线程机制。创建表对象时，可根据读写分离的原则，针对读、写各自创建所需的表对象，创建多表对象的代码如下所示：

```
...
int tbnum = 10;
String tbname = "student";
tbarray = new HTable[tbnum];
for (int i=0; i<tbnum; i++) {
   tbarray [i] = new HTable(hbaseconf, tbname);
}
...
```

每个对象可以对应于一个线程进行并行使用，在数据读取时，考虑到性能优化，需要设定如下参数，该参数的设定可在扫描数据时不用一条条传输，以提升效率：

```
tbarray[i].setScannerCaching(100);
```

当然，在执行查询时，也可以指定查询的列簇或字段信息，这样，可以减少数据量的传输，默认的 HBase 在集合查询时返回全部列簇数据。

在数据并行写的时候，可以考虑设置以下几个参数：

```
1  tbarray[i].setWriteBufferSize(10*1024*1024);
2  tbarray[i].setAutoFlush(false);
```

代码 1 处：设定写缓存，通过写缓存的设置，可以实现批量数据提交，以提高写效率。

代码 2 处：设定自动更新，为 true 时，每次提交即会更新，设置为 false 后，可以批量提交，避免每次提交带来的性能损耗。

2. 表池技术

表池类似于线程池，初始化创建一组表对象池，在使用时获取对象，使用后回收对象，以实现动态分配。

在 HBase 中，HTablePool 类定义了表池，使用 HTablePool 首先需要初始化对象，并且设定表池的大小；在使用时，通过 getTable 方法获取 HTableInterface 对象，并进行相应的数据存取操作，使用完后，通过 putTable 方法进行回收。

下面给出了一个表池的使用示例：

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.KeyValue;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.client.HTableInterface;
import org.apache.hadoop.hbase.client.HTablePool;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;
```

```
public class HTPoolTest {
    private Configuration hbaseconf = null;
    private HTablePool htbpool = null;
    //初始化
1   public void init() {
        hbaseconf = HBaseConfiguration.create();
        hbaseconf.set("hbase.zookeeper.quorum", "iZ23d4by1laZ");
        hbaseconf.set("hbase.zookeeper.property.clientPort", "2181");

        htbpool = new HTablePool(hbaseconf, 2);
    }
    //获取表对象
2   public HTableInterface gettable(String tbname) {
        return htbpool.getTable(tbname);
    }
    //回收表对象
3   public void puttable(HTableInterface htb) throws IOException {
        if(htb != null)
            htbpool.putTable(htb);
    }
    //获取某一行记录
    public void getrow(String tbname, String rowkey) throws IOException {
4       HTableInterface htb = this.gettable(tbname);
        Get get = new Get(Bytes.toBytes(rowkey));
        Result res = htb.get(get);
        System.out.println("family\tcolumn\tvalue\tversion\ttimstamp ");
        for (KeyValue kv : res.list()) {
            System.out.print(Bytes.toString(kv.getFamily()));
            System.out.print("\t" + Bytes.toString(kv.getQualifier()));
            System.out.print("\t" + Bytes.toString(kv.getValue()));
            System.out.print("\t" + kv.getMvccVersion());
            System.out.println("\t" + kv.getTimestamp());
        }
5       this.puttable(htb);
    }

    public static void main(String[] args) throws IOException {
        HTPoolTest ts = new HTPoolTest();
        ts.init();
        ts.getrow("student", "2");
    }
}
```

代码 1 处：为对表池对象初始化，参数 2 为初始的表池数量大小。

代码 2 处：通过表名获取表对象。

代码 3 处：回收表对象至表池。

代码 4 处：在使用表池时，获取表对象。

代码 5 处：使用完毕后回收表对象。

整个程序的运行效果如下所示：

```
[root@www hjava]# javac HTPoolTest.java
[root@www hjava]# java HTPoolTest
family  column  value  version  timstamp
info   student information  0  1459132909940
info  age  23  0  1459133055925
info  name  jim  0  1459135434239
```

7.4.4　批处理

HBase 支持针对数据增加、删除、查询、修改的批处理，借助批处理技术，能够极大地提高数据操作的性能。前面提到的 Put、Get、Delete、Scan 类继承于基类 Row，通过 List<Row>形成待操作的批处理集合，借助 HTable 类进行提交。

下面给出了一个批处理的使用示例，源代码如下所示：

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Delete;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Row;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.util.Bytes;

public class hbasebatch {

   public static void main(String[] args) throws Exception {
      //TODO Auto-generated method stub
      Configuration hbaseconf = HBaseConfiguration.create();
      hbaseconf.set("hbase.zookeeper.quorum", "iZ23d4by1laZ");
      hbaseconf.set("hbase.zookeeper.property.clientPort", "2181");

      Put put1 = new Put(Bytes.toBytes("3"));
      put1.add(Bytes.toBytes("info"),
        Bytes.toBytes("age"), Bytes.toBytes("12"));
      put1.add(Bytes.toBytes("info"),
        Bytes.toBytes("name"), Bytes.toBytes("tom"));

      Put put2 = new Put(Bytes.toBytes("4"));
      put2.add(Bytes.toBytes("info"),
        Bytes.toBytes("age"), Bytes.toBytes("42"));
      put2.add(Bytes.toBytes("info"),
        Bytes.toBytes("name"), Bytes.toBytes("alex"));

      Delete delete1 = new Delete(Bytes.toBytes("2"));

      //将增加、删除操作加入批处理列表中
1     List<Row> batchlist = new ArrayList<Row>();
2     batchlist.add(put1);
      batchlist.add(put2);
      batchlist.add(delete1);

      //让表加载批处理
      HTable htb = new HTable(hbaseconf, "student");
      Object[] res = new Object[3];
3     htb.batch(batchlist, res);

      //显示执行结果
      for(int i=0; i<res.length; i++) {
4        System.out.println(res[i]);
      }
   }
}
```

代码 1 处：实例化批处理列表。

代码 2 处：将数据操作加入批处理列表中。

代码 3 处：调用 HTable 的 batch 批处理执行方法。

代码 4 处：显示批处理操作日志。

整个程序运行效果如下所示，另外借助于 myhbase 验证查看运行后的数据结果：

```
[root@www hjava]# javac hbasebatch.java
[root@www hjava]# java hbasebatch
keyvalues=NONE
keyvalues=NONE
keyvalues=NONE
[root@www hjava]# java HBaseBase
myhbase:>scan student
row:3  info:age value:12 timestamp:1460037064286
row:3  info:name value:tom timestamp:1460037064286
row:4  info:age value:42 timestamp:1460037064286
row:4  info:name value:alex timestamp:1460037064286
```

7.4.5　数据迁移

对于传统关系型数据库的用户来讲，在使用 HBase 时，面临的最大问题是数据的迁移问题，即如何将关系数据库上的数据迁移至 HBase 中。这里给出两种方法，一种是基于 sqoop 工具实现数据迁移至 HBase 中，这种方法能够满足绝大部分数据迁移场景；另一种是编写专用的数据迁移程序，实现定制化的数据迁移，此方法只限于一些特定类型的数据迁移。

1. sqoop 数据迁移

sqoop 是 Apache 旗下的开源项目，主要应用于传统数据库(MySQL、Oracle 等)数据迁移至 HDFS、HBase、Hive 中，当然，也可以实现将 Hive、HDFS 数据迁移至传统数据库。

从官方网站下载 sqoop 压缩包(http://apache.fayea.com/sqoop/1.4.6/)，存放于 alidata 目录下，对压缩包进行解压，同时，对解压缩修改目录名为 sqoop，如下所示：

```
[root@www alidata]# tar zxvf sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz
[root@www alidata]# mv sqoop-1.4.6.bin__hadoop-2.0.4-alpha sqoop
```

这里以 MySQL 数据源为例，新建一个数据库 test_hbase，在里面新建一张表 staff，定义三个字段，并填充数据，相关的 SQL 脚本如下所示：

```
create database test_hbase;
use test_hbase;
CREATE TABLE IF NOT EXISTS `staff` (
  `id` int(11) DEFAULT NULL,
  `name` varchar(20) DEFAULT NULL,
  `age` int(11) DEFAULT NULL
) ENGINE=InnoDB;

INSERT INTO `staff` (`id`, `name`, `age`) VALUES
(1000, 'tom', 23),
(1001, 'jim', 23),
(1002, 'ttt', 24);
```

在 MySQL 中新建一个用户名 hbase，口令也是 hbase，用于数据迁移时使用，并赋予对库以及表的访问权限，如下所示：

```
mysql> insert into mysql.user(Host,User,Password) values("iZ23d4by1laZ
","hbase",password("hbase "));
mysql>grant all on test_hbase.* to hbase@'%'  identified by hbase;
mysql>flush privileges;
```

复制 mysql 连接包 mysql-connector-java-5.1.22-bin.jar 至 sqoop 的链接库中，如下所示：

```
[root@www bin]# cp mysql-connector-java-5.1.22-bin.jar /alidata/sqoop/lib/
```

利用 sqoop 测试连接 MySQL 数据库，并列出当前数据库 test_hbase 所有表，如下所示：

```
[root@www www]# cd /alidata/sqoop/bin/
[root@www bin]# ./sqoop list-tables --connect jdbc:mysql://
iZ23d4by1laZ:3306/test_hbase --username hbase --password hbase
...
staff
```

测试成功后，此时，可以通过 sqoop 将 staff 表的数据迁移至 HBase 中，在迁移命令时，在 HBase 中创建一个 staff 表，定义一个列簇 staffinfo，并以原表中的 id 字段作为 HBase 新表中的列键值，如下所示：

```
[root@www bin]#./sqoop import --connect jdbc:mysql:// iZ23d4by1laZ:3306/
test_hbase --username hbase --password hbase --table staff
--hbase-create-table --hbase-table staff --column-family staffinfo
--hbase-row-key id -m 1
```

此时，进入 HBase 查看数据迁移情况，如下所示：

```
hbase(main):007:0> scan 'staff'
ROW                          COLUMN+CELL
 1000            column=staffinfo:age, timestamp=1463038324515, value=23
 1000            column=staffinfo:name, timestamp=1463038212086, value=tom
 1001            column=staffinfo:age, timestamp=1463038329358, value=23
 1001            column=staffinfo:name, timestamp=1463038291049, value=jim
 1002            column=staffinfo:age, timestamp=1463038338395, value=24
 1002            column=staffinfo:name, timestamp=1463038299965, value=ttt
3 row(s) in 0.0210 seconds
```

2. 编程实现数据迁移

对于一些特定环境的数据，迁移有时候要借助于编程，即源数据需要处理，如前面的例子中，我们认为 staff 表的年龄信息是 10 年前上报的，在导入到 HBase 中时，每个员工的年龄要加 10，诸如此类的需求，单纯借助于 sqoop 就不太容易完成。这里通过编程的方式，实现前面的数据迁移，程序的主要流程是分别连接 MySQL 与 HBase，将 MySQL 数据表 staff 顺序读取，并以列的形式进行组织，写入到 HBase 中。

代码如下所示：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import java.sql.Statement;
public class DataMigration {

    //声明静态配置
    private static Configuration hbaseconf = null;
    static {
        Configuration testconf = new Configuration();
        testconf.set("hbase.zookeeper.quorum", "iZ23d4by1laZ");
        testconf.set("hbase.zookeeper.property.clientPort", "2181");
        hbaseconf = HBaseConfiguration.create(testconf);
    }

    //连接数据库
    public void DataMigr()
      throws SQLException, ClassNotFoundException, IOException {
        Class.forName("com.mysql.jdbc.Driver");
        Connection conn = DriverManager.getConnection(
          "jdbc:mysql://iZ23d4by1laZ:3306/test_hbase", "hbase", " hbase ");
        Statement stmt = conn.createStatement();
        ResultSet res = stmt.executeQuery("select * from staff");
        HTable htb = new HTable(hbaseconf, "staff_1");
        while(res.next()) {
            Put nameput = new Put(Bytes.toBytes(res.getString(1)));
            nameput.add(Bytes.toBytes("staffinfo"),
              Bytes.toBytes("name"), Bytes.toBytes(res.getString("name")));
            htb.put(nameput);
            Put ageput = new Put(Bytes.toBytes(res.getString(1)));
            int age = res.getInt("age") + 10;
            String strage = String.valueOf(age);
            ageput.add(Bytes.toBytes("staffinfo"),
              Bytes.toBytes("age"), Bytes.toBytes(strage));
            htb.put(ageput);
            System.out.println("insert [" + res.getInt(1) + "] OK!");
        }
    }
    public static void main(String[] args)
      throws ClassNotFoundException, SQLException, IOException {
        //TODO Auto-generated method stub
        DataMigration tt = new DataMigration();
        tt.DataMigr();
    }
}
```

编译运行结果如下所示：

```
[root@www hjava]# javac DataMigration.java
[root@www hjava]# java DataMigration
insert [1000] OK!
insert [1001] OK!
insert [1002] OK!
```

验证 HBase 中的 staff_1 表数据，如下所示：

```
hbase(main):011:0> scan 'staff_1'
ROW                           COLUMN+CELL
 1000          column=staffinfo:age, timestamp=1463109418494, value=33
 1000          column=staffinfo:name, timestamp=1463109418410, value=tom
 1001          column=staffinfo:age, timestamp=1463109418578, value=33
 1001          column=staffinfo:name, timestamp=1463109418539, value=jim
 1002          column=staffinfo:age, timestamp=1463109418664, value=34
 1002          column=staffinfo:name, timestamp=1463109418623, value=ttt
```

7.5 小　　结

本章梳理了 HBase 各编程接口的开发框架，以 rest 接口为例，介绍了如何通过 curl 与编程语言进行表、数据、集群的操作；以 thrift 接口为例，介绍了其编程流程；最后重点介绍了 Java API，通过安排一系列命名空间、表、数据的增删改查，以及过滤器、集群的监测与控制、表池与多表技术的编程示例，以加深读者的理解。通过上述示例的学习，读者应能初步掌握 HBase 开发，并能够在未来的项目中加以应用。

本章中安排了一个仿 HBase Shell 的 myhbase 示例，能够实现基于命令行的功能交互，有兴趣的读者可以去完善；此外，当前流行的管理是基于可视化特征的，也建议有兴趣的读者基于 Web 技术开发可视化的 HBase 管理应用。

大数据仓库篇

第 8 章

数据仓库概论

学习目标

本章首先介绍数据仓库的基本概念、特点，与传统数据库的差别，以期让读者对数据仓库有最基本的认识；之后，从业务角度来剖析出现数据仓库的原因；然后介绍数据仓库的体系结构，应用实例；最后，与读者共同探讨数据仓库这一领域的几个最核心的概念：ETL、数据质量、调度，它们也是构建数据仓库时不可或缺的组成部分。

通过本章的学习，读者应能够了解数据仓库的基本理论、核心概念和重要组成部分。

本章要点

- 数据仓库的概念和特点
- 数据仓库与数据库的差别
- 数据仓库主要应用于哪些场景
- 数据仓库常见的体系结构是什么样的
- 数据仓库里面的 ETL、数据质量、调度分别是做什么用的

8.1 初识数据仓库

近年来，大数据概念极度火爆，同时火起来的是互联网及相关行业的数据仓库。各大公司都在耗巨资招聘相关的人才、构建一整套数据仓库架构，并做持续的运营、管理和优化。我们都知道，商业的目标是利润，数据仓库为何会有这样巨大的商业价值？本节我们就来管中窥豹，看看什么是数据仓库，为什么需要数据仓库，它有什么用途。

8.1.1 什么是数据仓库

仓库是一个出现了上千年的概念，意思是盛装物品的巨大容器，比如粮食仓库、弹药仓库、原材料仓库等。而之所以称为“数据仓库”，也正是使用了仓库的本意。

顾名思义：数据仓库就是盛装了大量数据的巨大容器。

这里，我们一定要知道“数据仓库之父”William H.Inmon，他在 1993 年所写的《Building the Data Warehouse》一书中，定义了数据仓库的概念。

(1) 定义 1：数据仓库是一个面向主题的(subject-oriented)、集成的(integrate)、相对稳定的(non-volatile)、反映历史变化的(time variant)的数据集合，用于支持管理决策。

根据这一定义，我们看到数据仓库具有以下 4 个特点：

- 数据仓库中的数据是按照面向主题的方式组织的。
- 数据仓库中的数据是集成的。
- 数据仓库中的数据是稳定的。
- 数据仓库中的数据是随着时间不断变化的。

① 面向主题的。

正如我们上面举的例子：粮食仓库、弹药仓库、原材料仓库是不同的仓库，数据仓库同样也是按照其存放数据的内容不同，来分别存储和管理。

通常来说，数据仓库的主题是在较高的业务概念领域对整体的数据内容的划分，在划分的时候，一般不会苛刻地要求其准确性和完备性，而更像是一个业务概念，且更注重这个业务概念在逻辑上、在业务表现和分析方式方面的易用性。

比如，一家电商公司，核心业务是通过各种渠道进货；在网上销售给普通的用户；希望在其网站上做推广，以提升其产品销量。对于这样的客户，可以有单独的系统做销售方案配置；同时具有自己的售后体系。这家公司在做数据仓库主题划分的时候，一种选择就是将所有的业务划分为这样几个数据仓库：渠道数据仓库、用户数据仓库、客户数据仓库、售后数据仓库。每一部分都是独立的数据结构、独立的存储和分析应用。

请注意：在这个例子中，我们只是举了一个面向主题划分的例子。我们并没有说这种方案是完美的，甚至都不敢说是一定适用的，因为，数据仓库具有其阶段性和延展性，在后面会详细说明。

② 集成的。

粮仓的管理，是将一家一户的粮食都收集上来，做晾晒、防虫处理，之后按照粮食的类别存放在仓库的不同位置。数据仓库与此类似，它是要对主题相关的来自不同信息系统(或

者散落在个人手头、人脑等)、不同介质、不同存储结构、不同地域的数据加以收集、整理、分类和优化，之后才能够放入数据仓库中。

集成是一个复杂的体系，过程中也涉及大量的沟通、计算、管理工作。

③　相对稳定的。

对于粮仓来说，它存储的粮食，一般是为了储备，或者国家意义上的调拨和使用，一般不会今天拿两斤玉米、明天取 3 斤红豆。数据仓库也类似，存储在里面的数据，一般不会进行数据的删、改操作，业务数据入库之后，也极少进行更新。大部分的操作是查询，通过查询来取得信息、定位问题、进行业务优化、做方向决策等。

④　反映历史变化的。

粮仓里面的粮食都会被明确标注：粮食的收获日期、重量等。数据仓库也会不断地从业务库、日志数据等多种来源获取数据，标注它的生成时间，并把它们放入仓库中。数据积累的多了，整个数据仓库就能反映出每部分数据的历史变化。历史变化的情况指明了业务的变迁(粮食产量的变迁)，指导现阶段的健康指标(今年粮食产量是否合理)，同时，也可以用来预测未来的业务指标(未来一段时间粮食产量的预期)。

上面我们讲述了 Inmon 对数据仓库的定义，他是从数据仓库的特征给出的形式化定义。但是从上面粮仓的例子，聪明的读者可能会发现这里有个概念讲得很模糊。粮仓是装粮食的容器，而上面讲的数据仓库的定义描述的不是装数据的容器，反而讲的是数据内容。

近些年，在互联网业界，关于“数据仓库”，确实是每天都在做的事情，但并没有统一的、规范的定义。实际上，数据仓库是一个庞大的体系，需要很多人为了一个共同的目标去完成不同类别的事情，才能构建好一个完整、清晰、规范、业务有用的数据仓库。这其中的事情，至少可以分为两类：

- 针对数据内容进行梳理、协调、规范、统一。
- 完成数据处理、存储、展示等工具。

前者关注的是数据内容的准确性、可以理解性；后者关注的是数据操作的效率、管理上的易用。针对这样的一类区分，我们可以给数据仓库继续如下的定义。

(2)　定义 2：数据仓库是企业中针对海量数据的管理机制及其相关工具平台的总称。

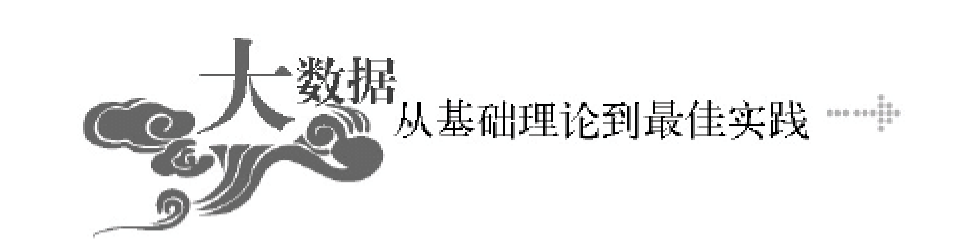
本书给出的这个定义对于数据仓库的实施更有指导意义。它明确指出，要构建数据仓库，必须有人去做数据管理，而且要有机制；还要有人去做相关的平台和工具。

定义中给出的两个方面互相依存、互不可以缺少：没有数据内容的管理，数据工具平台就没有价值；没有了工具平台，数据内容就会陷入一片混乱。一般来说，不论要做多大规模的数据仓库，这两方面都是齐头并进，相生相息的。

这里的定义，只是为了方便读者的理解和工作实践的思考。本书后面的内容中，不会特别去关注和提示所述是定义中的哪一部分，感兴趣的读者可以自行揣摩和梳理。

8.1.2　数据仓库与数据库

在大数据出现之前，数据库在传统应用中，占据了十分重要的地位——所有业务数据的存储都使用数据库，因此也出现了 MySQL、Access、DB2、Oracle、SQL Server 等很多数据库和厂商。其业务应用的结构一般如图 8-1 所示。



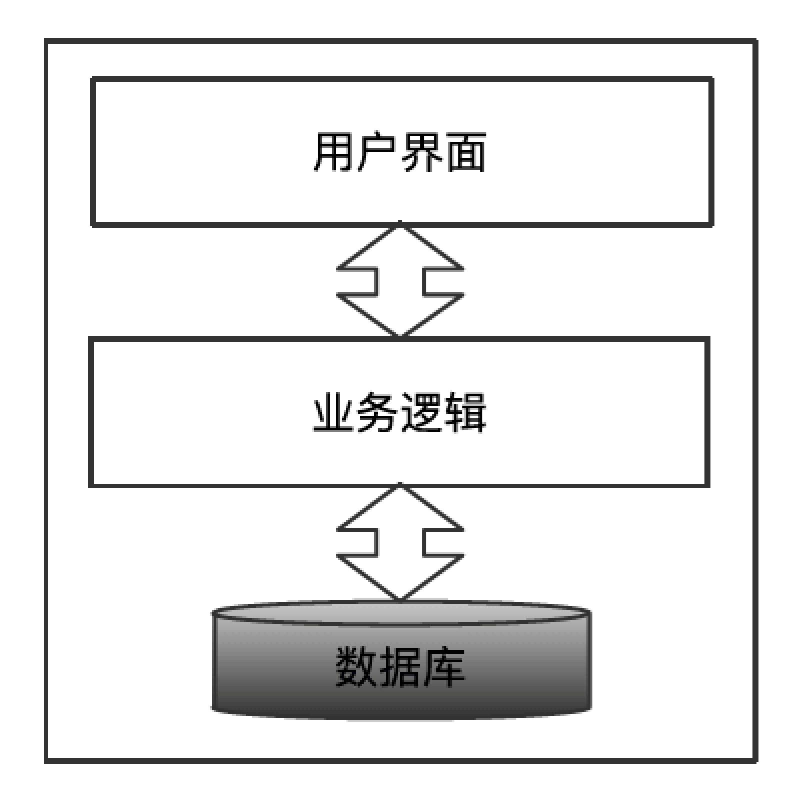


图 8-1　数据仓库出现之前的业务结构

而随着企业数据量的增大，为了更多数据的存储和分析的需要，数据仓库出现了，变成了如图 8-2 所示的结构。

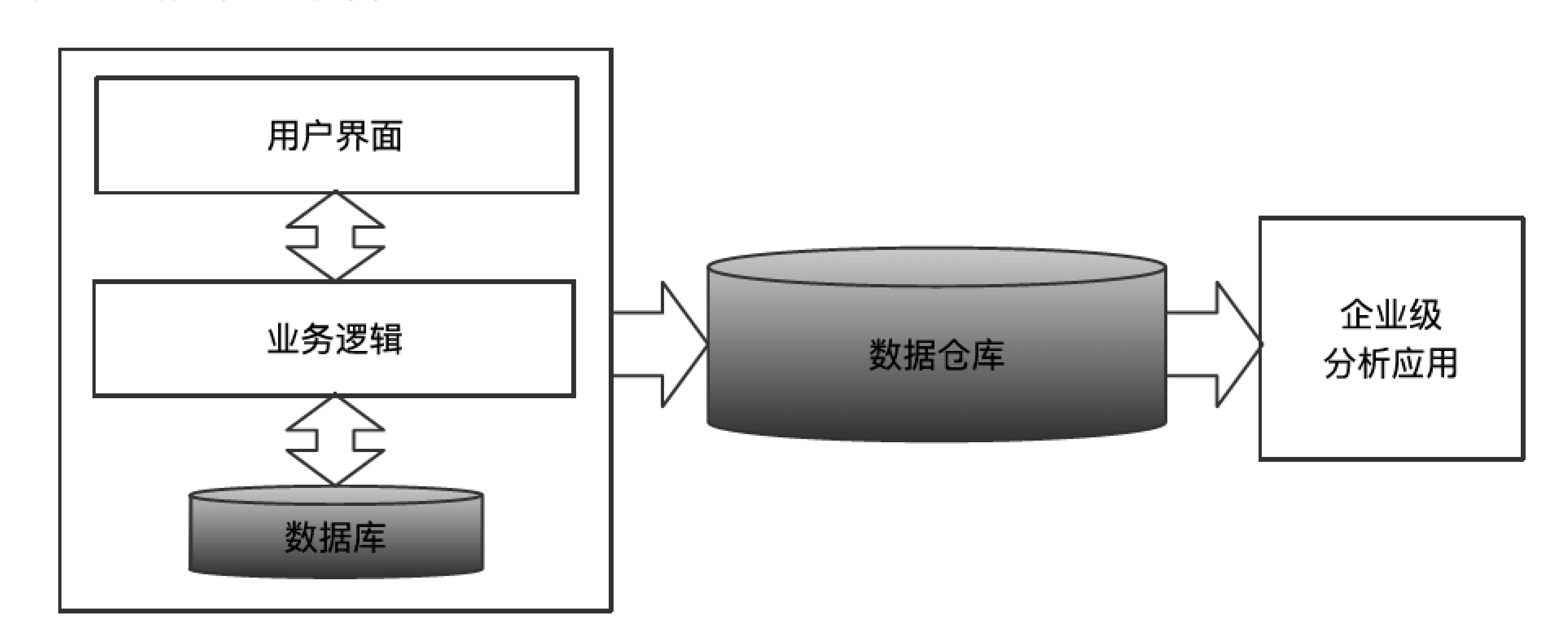


图 8-2　有了数据仓库后的业务结构

数据库和数据仓库的相同点是：它们都是数据的存储方式。那它们有什么不同呢？

1. 存储的内容不同

在传统的业务应用中，业务逻辑会随时来获取数据和修改数据，数据库里面需要时刻保持自己的数据是最新的(有一些数据库技术是用来保证毫秒级主从同步或多从库的状态一致)。而数据仓库中存储的是非常多的历史数据，如日志数据、业务应用的快照数据、工作组数据等。

2. 用途不同

一套数据库一般是一套业务应用的存储介质，用来提供业务需要的数据。而数据仓库一般是使用大量的历史数据、相关维度、相关工具等一起来做分析工作，这种分析工作，可能会作为管理者决策支持的依赖，可能会是业务模式修改的依据、也可能是业务系统修改和变更的考量。

3. 存取速度要求不同

数据库的存取速度直接影响到业务系统的访问速度，进而影响用户体验和业务表现，所以，数据库的存取速度一般要求很高(大部分都在毫秒级)。而分析工作一般会持续较长的

一段时间，因此，这个分析工作是可以接受分钟级甚至几十分钟的延迟的。

4. 存储的数据量不同

因为数据仓库存储了日志、业务系统快照等很多数据，所以它的数据量一般比数据库大很多。常见的业务应用数据库的存储量在 MB 到几百 MB 之间，而数据仓库一般在几百 GB 到几百 TB 甚至几十 PB 级别。随着大数据领域的发展，这个单位几年之后还会再经历几个翻翻。

5. 存在时间不同

数据库里面只保存最新的业务数据，只要有新的请求过来，旧的数据就不复存在了。而在数据仓库中，数据一般是会保存数月至数年(一般由于审计需要，企业数据仓库至少要保存一年以上)。对于现在的大型 IT 公司来说，一般会把自成立起就依赖的所有数据当作资产来保存，以备未来不可预知的分析需要。

6. 存取方式不同

业务系统会有各种操作，所以数据库的操作一般会包括增删查改，这种操作一般都是记录级的细小操作，而且并发度会很高(比如每秒几百次)。

而我们看到在图 8-2 中，数据仓库的数据流向是单向的，业务系统中的数据会进入数据仓库，这是批量加载的操作；之后，数据仓库的数据会流向企业级分析应用。批量加载数据不会修改历史，而只会批量地增加数据，这种操作一般不会十分频繁，常见的是按小时、按天地来加载。对于分析应用来说，它们只会从数据仓库中读取数据来做各种模型，但不会修改数据，这种读取操作可能次数会稍多(但也不会像数据库操作那么多)。

8.1.3　为什么要有数据仓库

在大规模数据出现之前，业务分析是通过数据库实现的。在互联网时代之前，公司间的 IT 系统是割裂的。每一家公司的系统只管理自己的业务：产品数据、客户数据、订单数据等都存放在数据中，表现形式是不同的表。由于业务规模小，日常的报表都只需要从数据库里面执行一个 SQL 就能完成，对未来业务规模的预期也完全依照管理者的经验。

随着 IT 基础设施的普及，和企业规模的扩大，一个数据库已经不能满足业务上的需要，可能需要多套面向不同业务的 IT 系统，因此，管理者也不能每天都通过执行一个 SQL 来完成例行的报表工作。

业务基础架构，根据不同的模式，会有不同的设计。但对于每一家公司来说，都需要根据已有数据，来做分析和决策，这时候，他们就需要拿到海量的数据，经过非常多的处理、整合，来形成一套可理解的架构，进而才能产出报表、分析报告、易于理解的数据产品、数据挖掘得到的策略等。这套数据架构和里面的数据，就是数据仓库，数据仓库是最先在大型企业中兴起的，因此，我们经常会听到“企业级数据仓库”这个名称。

这里我们也注意到，每家企业都会有基于数据仓库的报表、分析、挖掘等需求，所以在抛开数据内容的差异后，我们会发现，数据仓库的架构会有类似的地方。因此也就有了传统行业数据仓库厂商，如 Teradata、Oracle、IBM 等，这些厂商会提供面向行业的通用的数据基础架构。对于一家企业来说，在有限的人力和资金条件下，可以选择一款适合的产

品，从而使自己具有数据驾驭能力，能够让业务从数据提取的信息中受益。

随着互联网的普及，信息大爆炸时代到来了，出现了服务数百万、数亿人的大型网站，因此，新的业务模式也不断涌现出来：门户网站、搜索引擎、即时通信、电子商务、O2O(Online To Offline)等。由于互联网企业迅猛的发展，传统数据仓库的产品很难覆盖到如此快速的变化，因此，大型互联网企业都拥有自己的数据团队，他们会负责数据仓库的技术架构和数据仓库内容的梳理。

本书后面所介绍的数据仓库的内容，都是基于中国互联网行业近几年的实施经验做的总结和梳理。

8.2　数据仓库的核心概念

如果你关注数据领域，经常会听到很多与数据仓库相关的名词，听起来都是似懂非懂的。本节将与读者一起，认识一下这些名词，从概念的角度，来看看它们与数据仓库是什么关系，或者是数据仓库的哪些组成部分。由于大部分概念在业界很少有公认的定义，甚至不同公司在相关工作的职责和定位上都很不相同，所以我们尽量从易于理解的层面，来整理相关的概念。

8.2.1　数据平台

数据平台(业界有人会简称为 DMP，Data Manipulation Platform)，顾名思义，就是数据相关的平台，它是指完成数据生成、数据传输、数据存储、数据建模、数据分析、数据展示等一项或多项工作的一套系统。

不同企业的数据平台，其架构设计、实现方式、功能点都会有很大的差异，完全是依照企业发展的态势、相关技术架构和业务需求等多方面协调之后的结果。但大部分数据平台都会提供个性化定制数据服务的特点。

数据仓库重点关注海量数据的存储方式、模型及相关工具。如果狭义地理解，可以把它作为数据平台中数据存储、数据建模这一部分。业界对于数据平台和数据仓库这两个概念的理解和管理模式，一般会有下面两种。

一种常见的管理模式是把数据平台作为一整套基础架构，而数据仓库作为其中间处理的一个环节，从而形成一整套体系。在这种方案中，数据平台和数据仓库是一整套体系，它们需要遵循相同的规范、流程，即数据仓库是整个数据平台的一个环节。这种模式的好处，是仓库属于平台的一个部分，接口一致，易于管理，易于追查问题；弊端是数据仓库的实施可能会受限于数据平台整体的框架，扩展性不够；数据仓库工具的定位和实施也会牵动数据平台的架构变化。

另一种模式是让数据平台完成数据传输和存储的基础功能，最上层是数据产品的实施，中间层的数据处理过程交给数据仓库来做。在这种方式中，数据仓库只需要关注中间层数据流，可以有自己的架构和处理模式，因此，数据仓库架构和实施理论都是独立的，只需要面向下层做好数据入口、面向上层做好数据出口。这种模式的好处是，数据仓库独立成为一整套体系，可以根据需求不断地更新仓库的结构，易于扩展；弊端是仓库团队很难为

数据源的准确性和实施方案出谋划策，业务使用和数据生成部分会有脱节的情况。

以上两种模式的对比如图 8-3 所示。

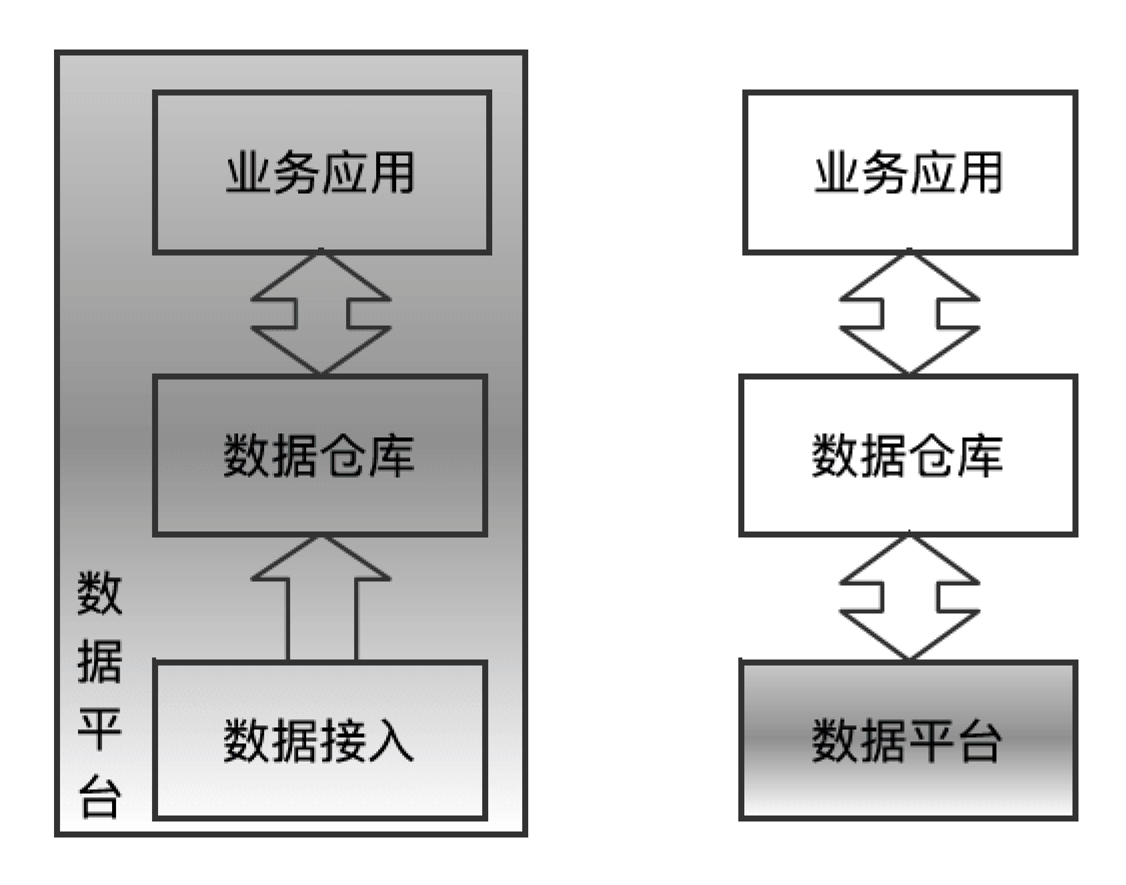


图 8-3　两种模式的对比

目前的互联网企业都会把数据平台和数据仓库的工作放在一个大的团队中来做(叫作数据仓库团队或数据平台团队)，并且会与业务团队和产品开发团队有密切的合作；但是，采用哪种模式一般不太固定，会根据情况选择一种时下适合的机制。

8.2.2　数据产品

在数据平台和数据仓库出现之前，就已经有数据产品了。比如国家统计局会提供各个行业的发展报告，比如天气预报，比如蔬菜价格行情等。它们都是把原始的数据加以包装，形成一个易于理解和方便使用的文件、服务、展演等。这就是广义的数据产品。

狭义的数据产品是指：组织内将原始数据经过数据平台或者数据仓库的处理后，形成的有形的数据展示、提取、使用的平台。一般来说，数据产品都会有丰富的界面展示，有多种交互操作，能够进行复杂的业务分析。

本书后面所说的数据产品，都是指狭义的数据产品。

数据产品有对内的(只有公司内部员工可以访问)，也有对外的。对内的，有例如报表系统、分析模式集成、多维分析 OLAP 等。对外的，有例如淘宝的数据魔方、淘宝指数、百度风云榜、百度大数据+ 等。它们都是基于企业海量数据，经过建模和挖掘，给用户提供有价值的指导。

数据产品有免费的，也有整个产品或者部分功能是收费的，后者基于“通过数据盈利”的思路，也是数据价值的一个体现。当企业的数据建设发展到一定规模后，一般都会考虑通过数据价值来“变现”。

从表现形态上来看，数据产品有报表型产品(静态报表/Dashboard/即席查询)、多维分析产品(OLAP，后面会详细介绍)、定制型服务产品(根据业务需求来定制产品内容和功能)、智能型产品(通过数据挖掘，提供一些深度的业务解读和预测)、使能型产品(通过模型的植入和数据表现，提供方向性建议，或者直接影响产品)。

8.2.3 商务智能(BI)

商务智能和决策支持都是数据产品的范畴。

商务智能产品(Business Intelligence，BI)是针对某类商务场景，基于复杂数据模型和挖掘结果的，具备一些复杂商务逻辑的数据产品。一般来说，商务智能产品是指导一部分具体商务活动的产品。

决策支持产品(Decision Support System，DSS)是面向公司决策层的日常数据供应和分析支持的产品。一般来说，决策支持产品都具备公司整体业务状况、运营状况的数据展示。

8.2.4 元数据

简单地说，元数据就是关于数据的数据，可以理解为数据仓库的数据字典。正如我们在操作系统中要存储文件/目录的元数据：文件名、文件大小、文件类别、创建人、修改人、访问时间等，在数据仓库中也要记录这些信息。如果没有这些信息，数据仓库中的数据就变成了无意义的 0 和 1 了。

数据仓库的元数据包括：数据仓库的数据来源信息、数据模型信息、业务数据与数据仓库数据结构的映射、仓库数据的使用情况等。

这里举几个典型的数据仓库元数据的例子：

- 数据仓库表的结构。
- 数据仓库表的属性(如存储格式、备注等)。
- 数据仓库表的源数据(是从哪里来，或者怎么计算来的)。
- 从源数据到仓库表的映射关系(包括其中的处理逻辑)。
- 数据模型的说明(依赖关系等)。
- 生成/访问记录。

在一个具有良好设计的数据仓库系统中，大部分元数据是通过系统的锚点(是指记录元数据的数据仓库架构中的插入点，当数据流经锚点或者有访问锚点的请求时，元数据就会被自动记录)自动记录的，之后，个别无法自动化的地方是人工去输入的。

准确和完备的元数据信息，为数据仓库的规范化、可持续性打下了良好的基础，数据仓库的模型设计、开发实施、运维管理、问题排查以及升级改造都要依赖于这些元数据。元数据是数据仓库不可或缺的最重要的组成部分。

好的元数据是数据仓库项目成功的关键。如果希望搭建长期的数据仓库，为业务提供服务，那从一开调研，就要记录好每一个环节的元数据，而且后面的所有数据处理和平台化都需要充分考虑元数据的记录和管理。

业界还有一种更高级的实施方案，就是元数据驱动的问题探查和业务分析，就是通过元数据的状态来检查任务的执行，进而定位问题，并完成基础的、自动的数据分析。

8.2.5 OLAP

在线分析处理系统(Online Analytical Processing System，OLAP)与联机事务处理系统(Online Transactional Processing System，OLTP)相对应。OLAP 是数据仓库最重要的应用，

本书后面 8.4 节会重点介绍这部分知识。

8.2.6　ETL

ETL(Extract、Transform、Load)是数据仓库中连接多层次数据的数据处理步骤，也是完成数据仓库模型建设的基础，本书后面的 8.5 节会重点介绍这部分知识。

8.2.7　数据质量

建筑工程项目讲求施工质量，质量关乎生命，没有质量的工程是不可信赖的。在数据仓库中的施工质量主要体现在数据质量上。

数据质量是数据仓库中数据内容是否符合业务及数据流定义的衡量标准。即：最终提供给业务的数据需要符合业务的定义，中间的处理过程需要符合最初对数据处理方式的定义。只有具有良好数据质量的数据才是可信的，也才是对业务有意义的。

在数据仓库实施的过程中，会有诸多因素的影响，导致出现数据质量问题。比如，需求调研不到位，团队就无法知道真正的、确切的需求内涵；数据梳理遗漏，会导致最终处理结果不准确；中间处理过程不规范，会导致偶发的 Bug 等。因此，要保证数据质量的良好，在整个仓库建设的过程中，不论是数据内容，还是相关工具，都要有据可查、有法可依，产出的结果必须是稳定的、可靠的、经过校验的。

几年前，作者曾经亲身经历过国内一家非常大的公司的一个典型的数据质量问题。同一个数据接口为部门 A 和部门 B 的多个线上模块提供服务。但由于大部门之间各自都忙于自己的业务，很少有时间去特别关注整体的业务一致性和数据一致性。直到某一天，由于公共的数据接口出现问题，导致 B 部门的线上服务出现了几十分钟的故障，影响用户达几千万，影响收入达几百万。这个时候，所有相关方才坐在一起去做事件分析，去梳理数据的含义。

目前，各大公司都非常重视数据质量，有的公司甚至请数据仓库相关的工作人员制定计划，专门开辟几个季度甚至一两年的时间来提高数据质量，正是为了从一开始就想办法避免上面所说的问题。然而，数据质量是一个多方位、复杂的、系统性的问题，如果想彻底解决，除了在工程质量、工作规范等方面指定硬性要求外，还需要相关的人员具有数据质量的意识和持之以恒的对高数据质量的追求、对业务负责的态度，这样才能真正有效、长期地避免数据质量问题的发生。

8.3　数据仓库中的数据内容划分

前面我们在讲解数据平台的时候，看到了两种数据仓库位置的示意图。数据仓库上层对接业务，下层对接功能简捷的数据平台或者数据平台的数据接入部分。基于这种概念，我们在讨论数据仓库体系结构的时候，就只关注具有上下依赖的数据仓库这一部分。

8.3.1　多个数据仓库

对于中小组织来说，构建一个完整的、集成了组织内所有数据的数据仓库是一个好办

法。当有数据需要入库的时候，将数据放入这个数据仓库；当有数据需要分析的时候，直接从这个数据仓库来获取数据。

当组织变得越来越大(比如上万人或者更多)、业务越来越丰富、业务逻辑越来越复杂的时候，如果仍旧使用单独一个大型数据仓库，运转起来会非常吃力。这个时候，一般会根据业务线的不同，来选择建立多个不同的数据仓库。

比如，在图 8-4 的右侧，这个组织是根据旧有业务的逻辑结构，划分成了用户数据仓库、客户数据仓库、销售数据仓库，并将新业务的数据单独作为一个数据仓库。

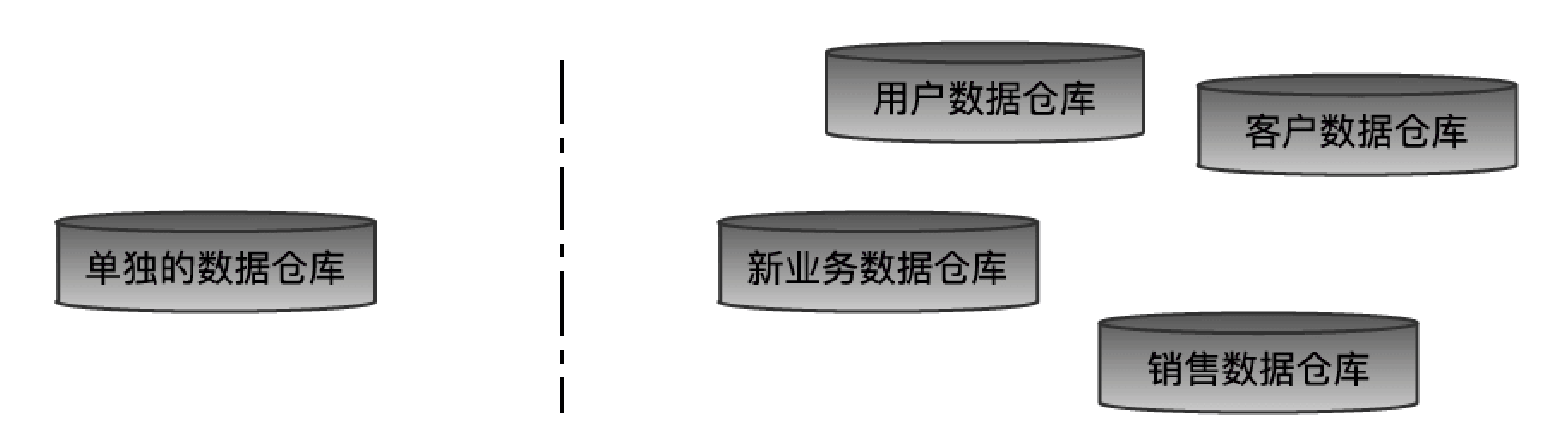


图 8-4　组织内一个和多个数据仓库的逻辑结构

大型组织内使用逻辑分离的多个数据仓库的好处如下。

1. 降低沟通代价

数据仓库的整个流程中，沟通工作一直贯穿其中，占据了所有工作的很大比重。比如数据调研、分层设计、元数据管理、运维等。在业务逻辑复杂、人员众多的场景下，沟通代价尤为突出。作者曾经历过，为了完成一项数据的统一，3、4 位数据产品经理在多个部门之间沟通了近一周，可以想象，如果把数据仓库中非常多的数据项都弄清意义、来源、口径并征得所有人员的认同，会是多么大的工作量。

在多个数据仓库架构中，组织内的数据产品只需要重点关注所在子数据仓库的所有数据，效率会提高很多。

2. 业务上的快速响应

正因为沟通代价降低了，相应的层次设计也会更明晰，功能开发和数据准备周期也会更短，所以业务上能够获得更快的响应速度。

3. 个性化支持

在单个数据仓库的大型组织内，要做任何数据修改、功能支持，都要牵一发而动全身，小改动都要评估大影响。而当只需要关注一部分业务的时候，可以做较多的个性化的事情。比如，客户数据仓库可以提供更复杂的客户分级逻辑和依赖此数据的管理工具，而不必担心这部分数据和工具因对其他数据内容的无效性而影响架构的问题。

另外，也可以根据业务规模大小、重要性高低等因素，来决定在不同的数据仓库上投入的人力、物理规模。比如，新业务属于探索性质，可能给予较少的人力来做比较基本的仓库支持。

上面我们关注了在大型组织内启用逻辑上的多个数据仓库的好处。我们可能会问这种方式是不是会有什么弊端？聪明的读者可能已经想到了，会不会造成 IT 基础架构的重复建

设？是的，这是一个在多个数据仓库架构下必须要考虑的问题。

这个问题如果解决不好，就会变成弊大于利的事情。成熟的大型互联网公司一般会有如图 8-5 所示的数据仓库 IT 架构。

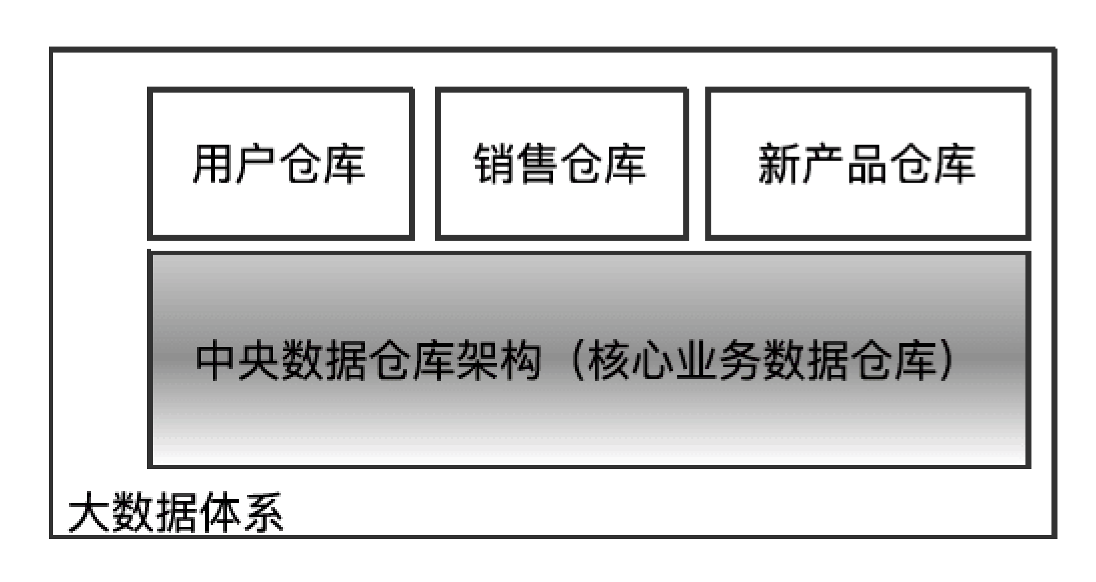


图 8-5　多个数据仓库的 IT 架构示意图

从中可以看到整个组织内的大数据体系，首先会有一个中央的数据仓库架构，会对各业务分支的数据仓库提供技术支持。有的时候，组织内最核心的业务数据仓库也是由中央数据仓库架构所在的部门来负责的。

在这种架构下，中央数据仓库架构的所有 IT 基础设施都会向上层的业务数据仓库团队开放。上层的仓库团队可以直接利用现有的统一的规范和平台来构建最基础的数据仓库数据，而最基础的数据，大部分时候，也已经在核心业务数据仓库中包含了。上层的各业务线大大节省了基础架构搭建开销和规范指定开销，以及一部分核心数据的整理和入库开销，它们只需要完成业务线最关注的那部分数据就可以了。

8.3.2　典型的数据仓库分层

在常见的数据仓库理论中，对于数据仓库中的数据分层是具有一致性认识的，分层结构如图 8-6 所示。

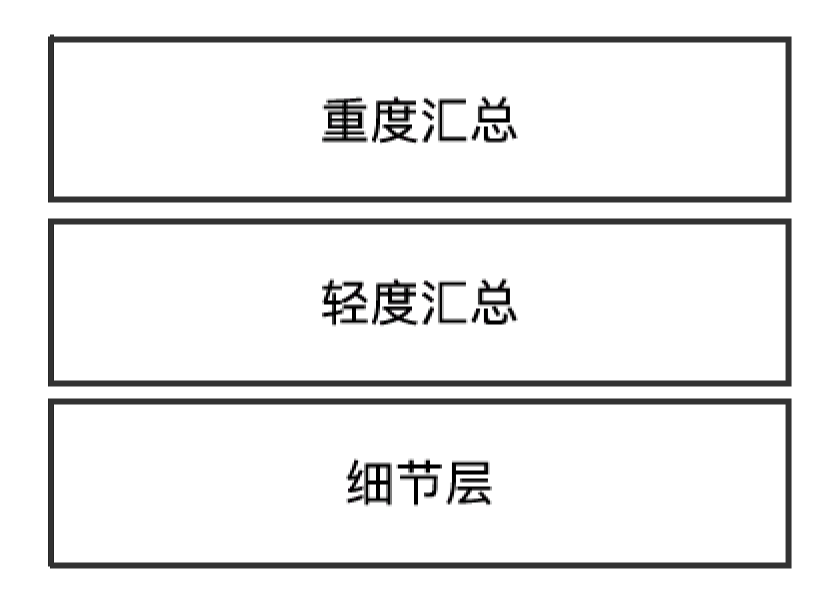


图 8-6　数据仓库的数据分层

1. 细节层

细节层又叫入库层，是外部数据入库后的第一个层次。

各大互联网公司最初使用数据仓库的时候都是保存日志数据，因为日志数据记录着用户的行为数据、业务状态数据和流转数据等，是做深度业务分析最直接的数据(基本业务分析可以直接从业务库里面查看数据)。同时，各大公司也从大量的日志数据里面获取了非常非常丰富的资源和巨大的业务价值，因此，细节层数据里面，最大量、最主要的数据是日

志数据。

细节层数据里面还有一类最重要的数据，是 ODS(Operational Data Store)。一般为了节省存储空间，节省网络开销，我们在产品端打日志的时候只会打印必需的字段，很多时候需要与业务库里面的数据关联到一起来分析，因此，业务库里面的数据也需要进入到数据仓库中，这部分数据就是 ODS。

当然，在真正实施数据仓库的时候，只有对于分析有益的数据才应该进入数据仓库中。而且要根据数据的种类不同，有不同的处理策略、机制和工具。但是，细节层的数据，大部分情况下，都会保留整个原始的数据，包括所有的数据行和所有字段(ODS 可能会只保留预期有用的字段)。

2. 轻度汇总

轻度汇总，是针对重度汇总来说的，顾名思义，就是只做一部分汇总。为什么不一次完成重度汇总呢？

在实际的数据使用场景中，一份细节层数据可能会被使用多次：在多个工具中，用来生成相同或者不同的汇总数据，或者在多次分析的时候都会获取。但是，它们对同一份细节数据的使用方式可能会不一样。这个时候，就需要综合各类需求，来形成一份能覆盖各类需求的中间层表。

这里一方面是为了做一套公共的汇总，尽量重用逻辑、减少数据，另一方面，可以在这个层次去关联重度汇总都需要的公共维度。

另外，在这里还需要考虑可扩展性，尽量把可预见的未来将会使用的字段都保留下来。

3. 重度汇总

重度汇总，又叫高度汇总。这个层次会直接提供上层的业务应用需要的数据。

小提示

① 细节数据 A 具有 9 个字段：a1、a2、a3、a4、a5、a6、a7、a8、a9。
② 工具 L 会使用 A 数据的 4 个字段：a1、a2、a3、a8。
③ 工具 M 会使用 A 数据的 4 个字段：a1、a2、a3、a6。
④ 分析师日常工作中常用 A 数据的字段：a1、a2、a4、a6。
⑤ 短期内 a5、a7、a9 字段没有使用场景。
⑥ 这时轻度汇总表 A1 就会保留 7 个字段：a1、a2、a3、a4、a6、a8。
⑦ 之后会按照工具 L、工具 M、分析师的需求来生成三张重度汇总表。

8.3.3 数据集市

数据集市(Data Mart)也是数据仓库领域一个特别常见的概念。

数据集市，也叫数据市场，是一个从操作的数据和其他的为某种特殊的专业人员团体服务的数据源中收集数据的仓库。

正如概念所述，数据集市也只是一个数据仓库，数据集市的特点是：

- 规模小，通常是面向部门的。

- 有特定的应用。
- 由业务部门定义、设计、开发和管理，响应快。
- 可升级到完整的数据仓库。

典型的数据仓库和数据集市的关系如图 8-7 所示。

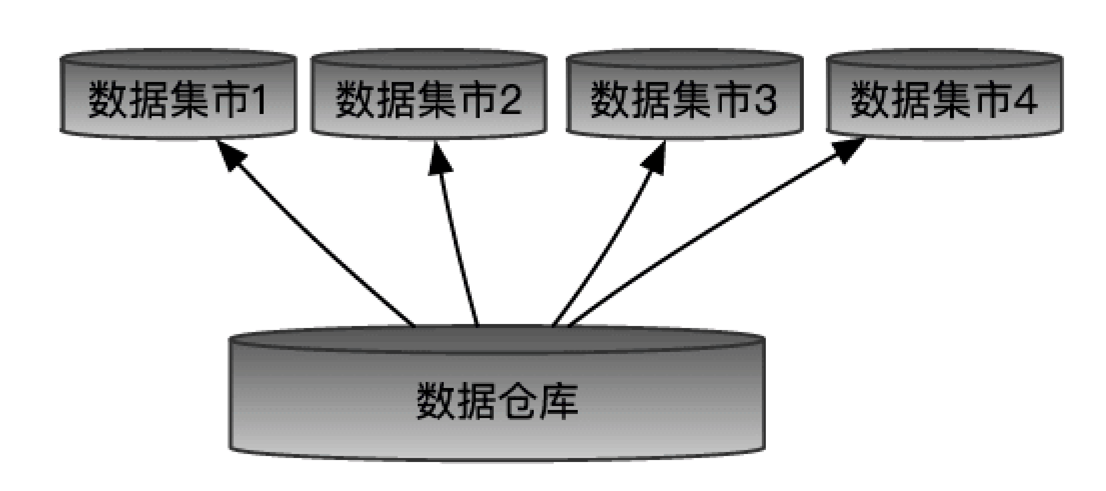


图 8-7　数据仓库和数据集市的关系

但是，不可否认的是，数据仓库和数据集市的属性是相同的，它们的不同点，只是规模和应用场景的区别。在组织内，数据仓库本身也有它自己发展的阶段性，所以，业界也不会将数据仓库和数据集市两个概念完全割裂开来看。

8.4　OLAP

在 20 世纪 60 年代，关系数据库之父 E.F.Codd 提出了关系模型，促进了 OLTP(OnLine Transaction Processing System，联机事务处理系统)的发展。1993 年，E.F.Codd 提出了 OLAP(OnLine Analytical Processing System，联机分析处理)，认为 OLTP 已经不能满足业务对数据的需求，计算机行业即将记入多维数据的时代。

OLAP 是数据仓库领域最著名、最通用的应用场景。任何数据仓库在构建的时候，都要考虑 OLAP 应用。

8.4.1　定义

OLAP 是基于对海量业务数据的理解和建模，系统化地为用户提供交互的多维分析服务的平台。

可以看到，OLAP 实际上是一个多维分析的系统，它与我们经常使用的 OLTP 有什么差别呢？表 8-1 做了详细的对比。

表 8-1　OLTP 和 OLAP 的区别

	OLTP	OLAP
系统用户	组织的外部用户	一般是组织内的业务分析人员
用户量	很多	很少
操作类别	读写	一般为只读
响应速度	毫秒，绝不超过 3 秒	秒级就可以接受，有的时间更长
后台数据	当前的、实时变化的数据	海量的历史数据
后台数据模型	常用实体关系模型	星型或者雪花模型

可以看到，相比于我们日常所用到的 OLTP 系统，比如电子商务网站(淘宝、一号店等)、门户网站(新浪、网易等)等，OLAP 更着重于给组织内的业务人员提供分析服务。

8.4.2 维度建模

1. 核心概念

在讲解多维建模前，我们先要知道如下几个概念。

度量(Metric，也称作指标)：是人们最想关注的最终结果数据。例如总收入。

维度(Dimension)：是人们对度量的最终值的拆解方式。例如，总收入的数据可以分地域、分季度、分用户类别来拆解，其中的地域、季度、客户类别就是维度。

多维数据：将度量值按照多个维度进行拆解，就形成了多维数据。比如总收入按照地域、季度、客户类别进行拆分，就可以形成一个三维的立方体。下面是一个示意图(图 8-8)，看起来像是一个 4×3×4 的立方体，因此，多维数据也叫多维数据立方体(Cube)。维度更多的时候，是不能用空间几何的概念去表示的，但是人们还是依旧延续使用了多维数据立方体这个名称。

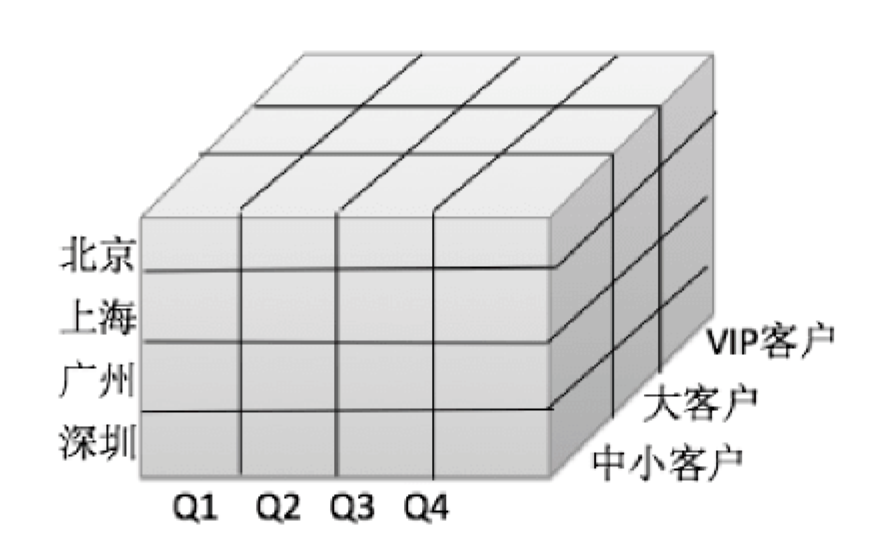


图 8-8　三维数据示意图

切面(Slice)：也叫切片，是指多维数据立方体中确定了一个维度之后得到的数据子集。例如，图 8-8 中，我们确定了地域维度的值为上海，则形成的数据子集就是二维的(Q1、Q2、Q3、Q4)*(VIP 客户、大客户、中小客户)。

切块(Dice)：是指在多维数据立方体中选择一个局部范围的数据块。一般可以选择一个或者多个维度的一个区间，或者任意的维值。例如，(北京、广州)*(Q2、Q3)*(中小客户)就是一个切块。

上卷(Roll-up)：是指在多维数据立方体中，将维值变换为它的父维值，从而获得汇总级的数据。比如，图 8-9 中，从右到左，取季度维值的父维值(全年)，就是一个上卷操作。这个操作经常发生在分析过程中暂时不关注一个维度的时候。

地域	收入（万）
北京	90
上海	80
广州	85
深圳	78

⇔

地域	收入（万）			
	Q1	Q2	Q3	Q4
北京	25	25	20	20
上海	20	20	20	20
广州	20	15	30	20
深圳	18	20	20	20

图 8-9　上卷和下钻示例

下钻(Drill-down)：是指在多维数据立方体中，将维值变换为它的子维值，从而获得细节级的数据。比如，在图 8-9 中，从左到右，取季度维值的子维值(Q1、Q2、Q3、Q4)，就是一个下钻操作。在多维数据分析时，上卷和下钻都是常见的操作。

2. 模型设计

数据模型是指对数据仓库中各个数据元素以及它们之间关系的描述。由于 OLAP 是数据仓库里面最具特色的应用，因此，数据仓库建模也叫维度建模(或者多维建模)。

OLAP 中有两类表：事实表和维度表，维度表就是对维度的描述，事实表就是多维立方体中的指标数据。在数据仓库多维建模中，有两种常用的模型设计方式：星形模型和雪花模型。图 8-10 就是上面举例的收入数据的星型模型，可以看到，星形模型的特点就是所有的维度都是单独的一张表，而且都是事实表直接依赖的维度表。

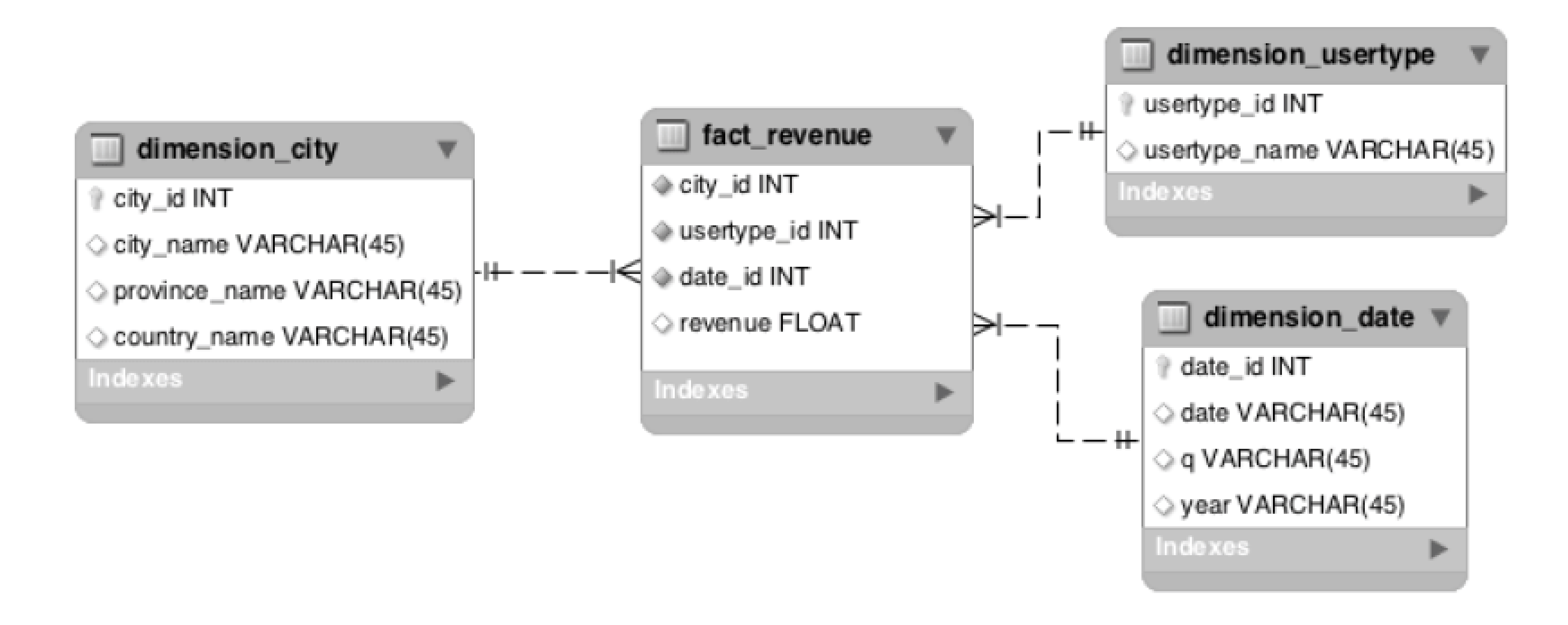


图 8-10　星型模型

通常，在我们的数据库课程里面会有范式的概念，在数据仓库领域依旧有效。我们观察一下上面的星型模式，是否还可以有更好的泛化程度的模型？图 8-11 是一个雪花模型，可以看到，雪花模型的特点是对于维度中的重复字段，采用单独的表进行关联。

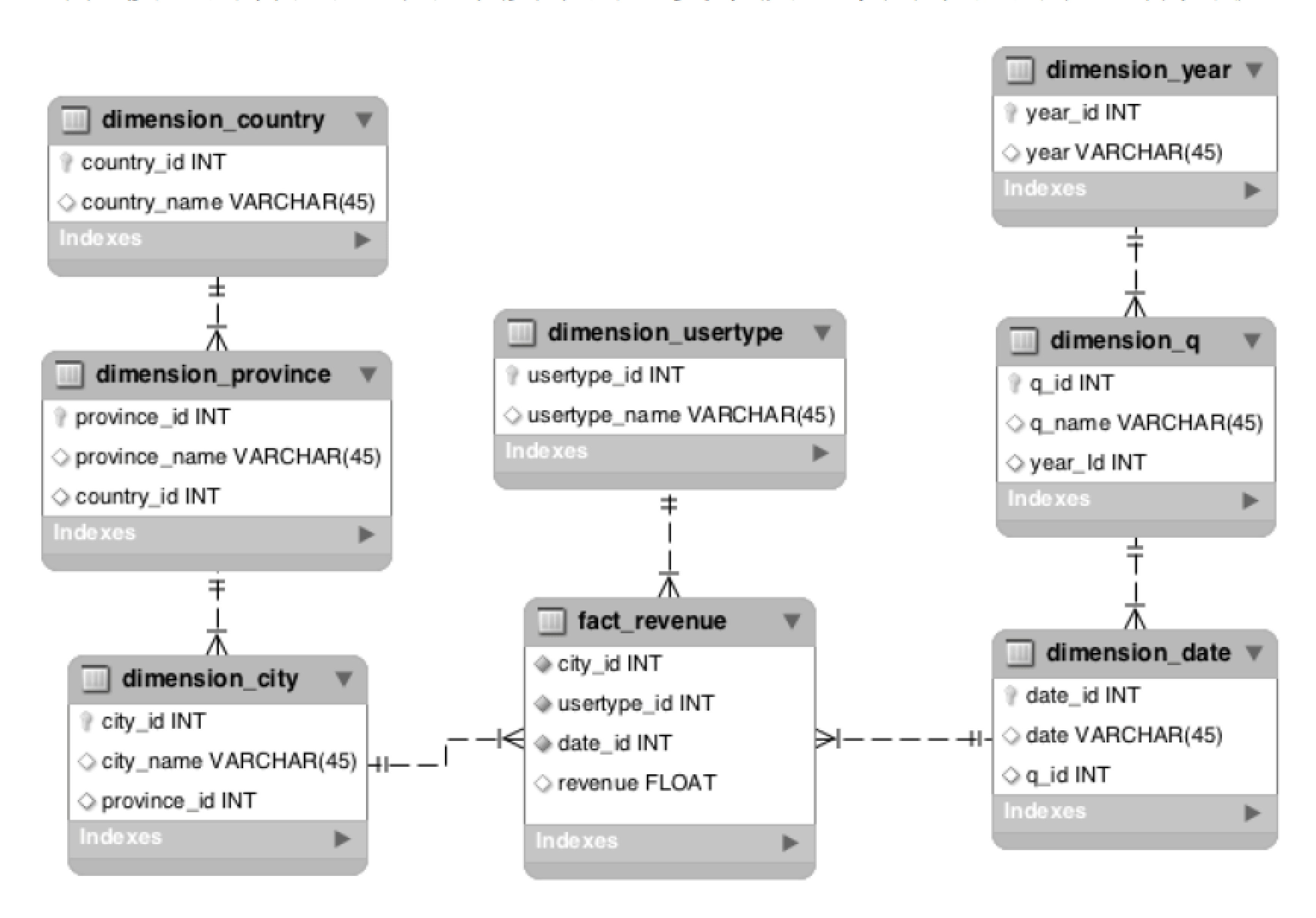


图 8-11　雪花模型

雪花模型更符合数据库设计中的范式要求，但是，在查询数据的时候，需要从事实表进行多次关联，来获取不同级别的维值。

星型模型和雪花模型各有利弊，需要根据业务的需要，以及查询引擎的状况选择合适的方案，并对不同的业务场景选择适合的方案。

小提示

上面只是为了讲解内容的方便设计的一个示例，并不一定是最好的设计，真正实施的时候，需要根据情况，来决定使用的模型和字段设计。

8.4.3 事实表

1. 指标有多种类别

在数据仓库中的指标有多种类别。可以按照是否允许累加对指标进行分类。

第一种最常见的就是可以直接累加的指标，上面的收入就是可以累加的指标。

第二种是不能累加的指标，例如消费客户百分比。

第三种是在某些维度上可以累加的指标，例如消费的客户数，在客户类别维度上可以累加(假设一个客户的类别是唯一的)，但是，在季度维度上不可累加。

可以按照指标值的种类进行分类：整数类指标、百分比类指标、小数类指标。

还可以从指标计算方式来分类：直接计算指标、间接计算指标(通过直接计算指标计算就可以得到)。

2. 事实表存储方式

我们看到指标有这么多种类别，那在事实表里面应该如何存储呢？本书不讨论复杂的系统设计，我们只关注一种最简单的可累加指标的存储。

粒度：数据仓库中保存数据的细化或者综合程度的级别。细化程度越高，粒度越小。

事实表的第一种存储方式就是只存储最细粒度的数据，在 OLAP 查询的时候实时汇总，比如，如图 8-12 所示的最细粒度的事实存储。

季度	客户类型	地域	收入（万）
Q1	VIP客户	北京	3
Q1	VIP客户	上海	2
Q1	VIP客户	广州	3
Q1	VIP客户	深圳	2.3
Q1	大客户	北京	2
……	……	……	……

图 8-12　最细粒度的事实存储

另一种存储方式就是在存储的时候做预汇总，这样，查询的时候直接取符合条件的记录，不存在实时计算环节，这种方式，在一定程度上能提高查询效率，可以给用户提供更快的服务。

图 8-13 给出了预汇总的事实存储的一个示例。

季度	客户类型	地域	收入（万）
Q1	VIP客户	北京	3
Q1	VIP客户	上海	2
Q1	VIP客户	广州	3
Q1	VIP客户	深圳	2.3
Q1	VIP客户		10.3
Q1	大客户	北京	2
……	……	……	……

图 8-13　预汇总的事实存储

8.4.4　维度表

一种最简单的维度就是一旦出现，则不再改变，比如日期和时间(当然，我们不讨论复杂业务场景中对日期和时间的各种划分问题)。

在实际数据仓库建设的过程中，大部分维度都会存在更改的可能。如果频繁更改(比如每个计算周期都会有大比例修改)，则直接使用新的维值来替代旧有的维值。比较复杂的一种情况是缓慢变化维(Slowly Changing Dimension，SCD)。

例如，上面所说的客户类型，当一个客户刚刚进入系统时，他是一个中小客户，随着客户的不断成长，可能会在一段时间以后变为大客户，进而再变成 VIP 客户。对于这种情况，如果我们的客户信息维度表里面只存储最新的信息，当存在历史数据回溯的时候，就会导致客户类别错乱。

对于缓慢变化维，一般有如下几种使用办法：

- 只使用最新的维值。在有些特殊的业务场景下，简单处理就能满足需求。
- 维值也保存所有历史记录。
- 记录维值生效的时间段。在使用维值数据的时候，根据时间来获取符合条件的，例如，如图 8-14 所示的缓慢变化维记录起止时间段。

客户	类别	开始时间	结束时间
Custom1	中小客户	20120109	20140203
Custom1	大客户	20140204	20150505
Custom1	VIP客户	20150506	
……	……	……	……

图 8-14　缓慢变化维记录起止时间段

本节讲解了数据仓库中 OLAP 和建模相关的知识，这些是直接面向上层应用的，那么，从原始的数据如何一步一步地形成 OLAP 模型呢，下一节我们来详细学习。

数据仓库本身也有它自己发展的阶段性，所以业界也不会将数据仓库和数据集市两个概念完全割裂开来看。

8.5　ETL

ETL 是数据抽取(Extract)、转换(Transform)、加载(Load)的简称，是指从输入源获取数据，把数据转换成合适的格式并加载到目标数据库中，是数据仓库中数据流转的主要方式。

上一节我们了解到，数据仓库会根据数据的汇总程度和用途来细分为多个层次，一般

来说，仓库中的数据，都是来自于数据源，经过 ETL 的过程进入仓库的细节层，然后经过 ETL 过程完成轻度汇总，再经过 ETL 完成重度汇总，从而把数据提供给数据应用方。在整个数据流动过程的每个阶段，都是“输入”经过 ETL 处理之后得到“输出”的过程。ETL 在数据仓库层次中的位置如图 8-15 所示。

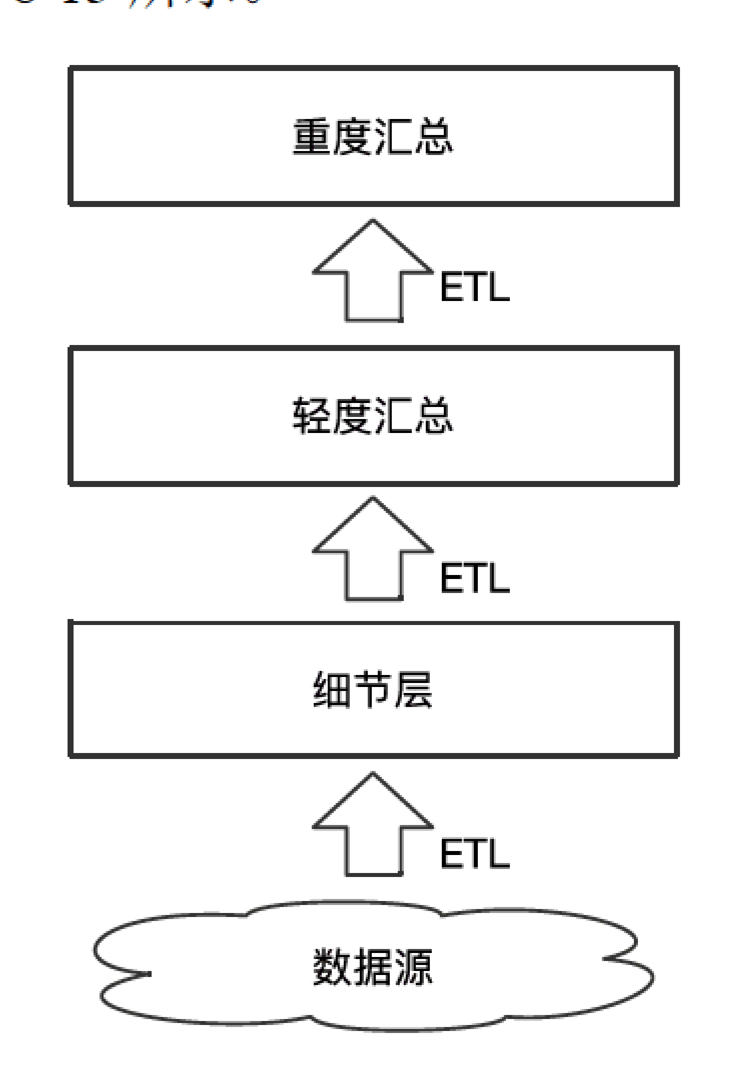


图 8-15　ETL 在数据仓库层次中的位置

从图 8-15 可以看出，数据仓库的分层结构指的是仓库中的数据，而 ETL 指的是数据在层次之间的流转。ETL 是数据仓库分层结构的实现方法，没有 ETL，就没有办法实现数据仓库的分层结构。

ETL 是数据仓库中最基础，也是最重要的工作，没有 ETL，任何设想和架构设计都是纸上谈兵。在大部分数据仓库建设的过程中，ETL 大都会占据工程实施工作 70%以上的工作量。

8.5.1　抽取

通常，数据仓库中的数据有多个数据来源，比如日志、各种业务系统、用户输入等。而不同来源的数据一般就会来自不同的位置、不同的存储架构和访问方式。因此，常用的抽取工具会根据业务需求的场景，来决定需要兼容哪些数据源和哪些抽取方式。

海量的日志一般存储在机器上或者 HDFS 上，这时的抽取一般通过 WGET 或者 MapReduce 的方式来获取数据。常见的业务数据是存储在 MySQL/Oracle 等传统数据库中的，这时，通常通过 JDBC 等方式来获取数据。对于有强用户输入需求的场景，一般会给用户提供输入接口或者可视化界面(当然，这种需求一般在上层的业务平台来满足)。

8.5.2　转换

1. 数据集成

从不同数据源获取的数据，会出现五花八门的情况：对于相同内容的描述不一致，对于相同内容的编码、命名也不一致，这时候，就需要完成数据集成的工作。

数据集成，是指对多种内容、多种存储方式的数据内容做规范化、归一化的过程。如图 8-16、图 8-17 所示的这些操作，就是常见的数据集成操作。

图 8-16　描述合并

图 8-17　编码规范化

2. 清洗

清洗是指通过一定的数据处理，得到符合规范要求数据的过程。

从仓库外的系统获取到的数据，通常不是能直接满足需要的数据，它们或者定义与业务需求不一致，或者多出了很多分析无用的业务字段，或者多出了很多分析无用的业务记录。这个时候，就需要经过一定的数据转换，来形成满足需求的数据。清洗的过程一般有三个阶段。

第一个阶段，是明确已有数据的范畴。这点是在 ETL 实施过程中经常被忽视的一点，被忽视的原因，通常是 ETL 工程师通过观察，自己给数据下了一个定义。人工对数据的观察会受限于观察者的知识结构和业务水平，也会受限于观测时间和数据片段的选取，由于 ETL 工程师不是业务系统专家，他通过观察数据，“得到的结论”常常被证明是片面的和考虑不周的。比如这几种情况：

- ETL 工程师观察到的数据片段对于所有记录的字段 C 都有值，因此在清洗的时候认为 C 字段必有值，实际上，数据会在一周内偶发 C 字段为空。
- ETL 工程师观察到的数据片段所有记录的汇总值恰好与需求方的要求完全一模一样，因此，在清洗的时候，认为这就是需求方要求的数据，实际上，冗余数据出现的业务场景恰好在这个数据片段内没有出现。

因此，明确数据范畴的时候，一定是经过相关的业务专家来详细确定数据的范畴，和细节的业务含义。这个阶段一般在数据调研的阶段完成一次，并签订“数据供给协议”(通常是组织内认可的有一定协议性质的文字)。

第二个阶段是将明确数据范畴的数据经过规则变换(清洗过程通常是指数据过滤规则)形成业务需求的数据。因为已经有了明确的数据范畴和详细的定义，此时的变换规则就能选择比较合适的，这里的变化规则也是历次数据供给都需要按照指定的方式来运行的。

第三个阶段是数据监测，即检查经过变换的数据是否符合清洗的预期。简单的数据监测，就是完成数据结果的总体和抽样检查，复杂的数据监测需要在历次数据供给之后都做

一次探针监测。当然，数据监测不是清洗所特有的，所有的数据转换和加载步骤都可以植入数据监测探针。

3. 其他转换

(1) 排序。

需要排序的场景是多种多样的，例如按照 Session 的访问序列来产出数据；例如需要按照某个指标的结果值提供头部数据。

(2) 合并。

把数据合并起来，例如，要获取总的收入，就需要把各种类型客户的数据合并到一起。

(3) 列剪枝。

大部分时候，我们只需要所有数据的一部分列来做分析，这时，就需要把无关列排除在外，这个过程叫作列剪枝。

(4) 汇总。

汇总就是形成数据仓库三级层次结构的基础。

(5) 累计。

累计是按照某个规则，生成从开始时间至今的所有数据的集合。通常，累计不是简单的合并。比如，需要从 2015.1.1 至今的所有客户的信息，这里的要求，就是每个客户的信息都是唯一的(而不是对 2015.1.1 至今的所有数据做汇总)。

(6) 关联。

按照某些条件，把两份数据关联到一起，以便于获取各自的部分字段。关联通常是指与维度数据做关联，以便于获取一级或者多级的维度值，从而得到汇总值。

8.5.3 加载

加载就是把数据装载到数据仓库的某一个层次中，以便于上级层次或者仓库用户使用。

1. 添加加载

添加加载是最常见的加载方式。一般组织内在搭建数据仓库后，需要持续地做数据仓库的管理和运维，并保存一段时间内的历史数据。添加加载就是将一个时间周期(比如天、周等)内的数据直接添加到数据仓库中。

添加加载的结果，就是数据在每个时间周期内都会有一份数据。

2. 覆盖加载

覆盖加载就是在加载的时候，把原来的数据清理掉，然后再装载新的数据。

覆盖加载的结果就是数据只保留一份。

3. 累计加载

本时间周期的累计数据的生成方式是：把上一个时间周期内的累计数据和本时间周期内的数据做累计。

累计加载的结果，是至少会保留最近的时间周期的数据。

4. 单次加载

顾名思义，单次加载就是只加载一次数据。也就只有一份数据(没有时间周期的概念)。

5. 数据初始化

上面说的四种加载方式，除了单次加载外，其他的都是周期性加载。对于周期性加载，当我们完成 ETL 工作后，一般会涉及到历史数据回溯。历史数据回溯也是 ETL 无法绕开的一个关键问题。

添加加载的历史数据回溯，只需要将需求时间段内的 ETL 都运行一遍即可。

覆盖加载只需要运行最新一个时间周期的数据即可。

累计加载的历史数据回溯，需要先初始化第一个时间周期的数据，这个过程一般会单独生成数据的内容；之后，再把需求的时间段内的所有 ETL 按照时间顺序运行一遍。

8.5.4　ETL 元数据

前面的章节中，我们讲到过元数据在数据仓库中非常重要，在 ETL 的过程中，数据过程和结果的元数据正是元数据的主要内容。

1. ETL 过程的元数据

ETL 过程的元数据是指在抽取、转换、加载的过程中任务处理逻辑、运行基本情况等的说明。具体如：

- ETL 的逻辑语义描述。
- ETL 的主要数据源和结果概述。
- ETL 结果和中间字段的计算方式详述。
- ETL 结果和中间字段的逻辑负责人、上游接口人。
- ETL 在每个时间周期内的运行起止时间、资源占用情况、数据处理情况。

2. ETL 结果的元数据

ETL 结果的元数据是指加载到仓库中的数据结果的描述。具体如：

- 数据整体内容的逻辑语义描述。
- 数据整体内容的来源概述。
- 数据在每个时间周期的数据量、数据位置、数据抽样、保留时间。
- 数据在每个时间周期的数据特性。
- 数据在每个时间周期的访问次数。
- 数据的各字段类型、字段含义、是否大小写敏感、是否允许为空。
- 数据生成的负责人。
- 数据的所有人。

3. 数据质量的元数据

如果组织内对数据质量有比较严格的要求，就会在整个数据处理的各个环节增加数据质量探针(探针是指检查数据质量的小部件)。则对于 ETL 的整个过程，以及结果数据，都

可以增加数据质量的监测，并将结果纳入元数据管理的范畴。

数据质量的元数据，是保证需求高质量完成的重要环节，可以有效地避免人工检查和开发测试阶段未发现问题引起的业务误判。具体如：

- 检查数据是否有错误。
- 检查数据是否完整。
- 检查具有业务含义的一些关键指标是否符合预期，并通过质量探针持续地观察和分析。

可以看出，数据质量的监测，能够尽早地发现影响业务数据的问题，而不是等人工去检查 ETL 结果元数据的时候才发现。

8.5.5　ETL 工具

当完成了数据仓库的设计，了解了 ETL 的含义和需要做的工作概貌后，就需要完成 ETL 过程了。ETL 对于大多数组织的仓库建设来说，都是具有一定工程量的工作。

由于对于相同类别数据的访问具备一定的相似性，而且数据仓库的架构又具有统一性，这时候，如果有一个很好用的 ETL 工具，则可以避免过多的重复性的编码工作。

最常见的 ETL 工具是通过配置项来完成 ETL 过程的，比如配置数据源、配置需要的字段、配置转换过程、配置加载位置，之后，再配置运行规则，这个时候，一个 ETL 任务就开发完成了。

未来，如果有 ETL 工具的升级，则可以兼容先前的配置规则，或者批量修改配置，都不需要单个 ETL 任务去处理。

还有一种更高端的 ETL 工具，是在可视化界面中完成上面所述的所有配置，看起来会更简单明了，而且相关的工作还可以交给非 ETL 工程师的人来辅助，一起推进工作更快地完成。一般来说，这种 ETL 工具都会与后面所介绍的调度在同一个平台上提供支持。

对于没有类似基础设施的组织，可以根据组织的情况，选择购买成熟的解决方案，或者调研开源方案，并做个性化的增量开发来解决(由于相关的产品和开源框架都非常多，而且技术更新很快，本书暂不提供详细的推荐，避免误导读者)。

8.6　调度和运行

调度这个词汇，通常会出现在大批量运行时的管理场景，比如出租车调度，意思是管理和支配成千上万辆出租车的位置和接送人信息，以便于使出租车能更快地响应更多人的出行需求。

在数据仓库中，70%的工作是 ETL，我们这里所讲的调度，就是 ETL 任务的调度。设想一下：如果有 20 个大的需求类别，每个需求可以细分为 5 类有关联的数据，每类数据需要 10~20 个不等的 ETL 任务，这些任务需要按照它们的要求运行起来，这个时候，就有上千个任务需要管理，已经远远超出个人手工操作的范畴(而这个例子只是一个很基础的数据仓库，复杂的会更多)。因此，我们需要一个统管全局的调度。

8.6.1　调度怎么工作

对于一个 ETL 任务来说，要完成它的历史使命，需要有如下几个阶段：任务生成→接受调度→运行→完成。

因此，除了调度外，还需要有一个运行的功能。对于任务调度和运行系统来说，它的输入是外部给予的 ETL 任务，完成调度运行后，输出的是运行结果。

如果把调度和执行分开来看，任务是由调度逻辑调起，然后交给执行机器去运行的。当然，它们之间也会有任务状态的更新，如图 8-18 所示。ETL 的任务，就是数据仓库三个层次中很多的数据操作步骤的集合。

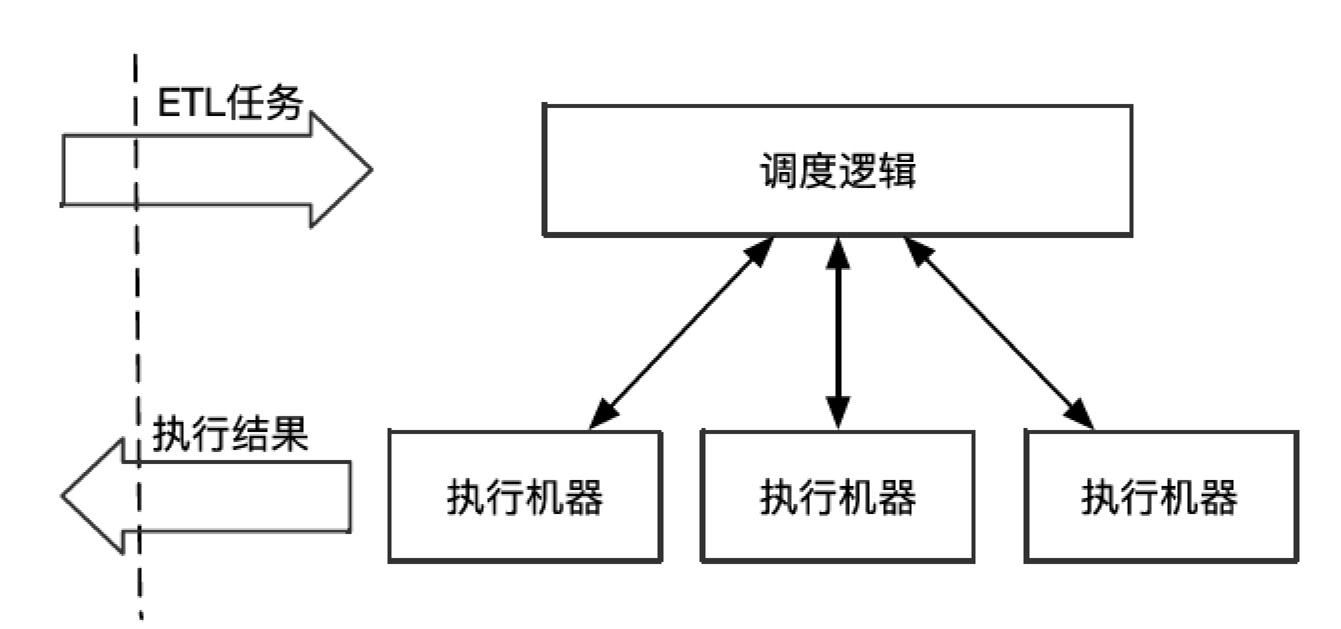


图 8-18　调度的工作方式

在一个 ETL 任务的生命周期中，一般会存在多种状态，它们之间的流转关系如图 8-19 所示。

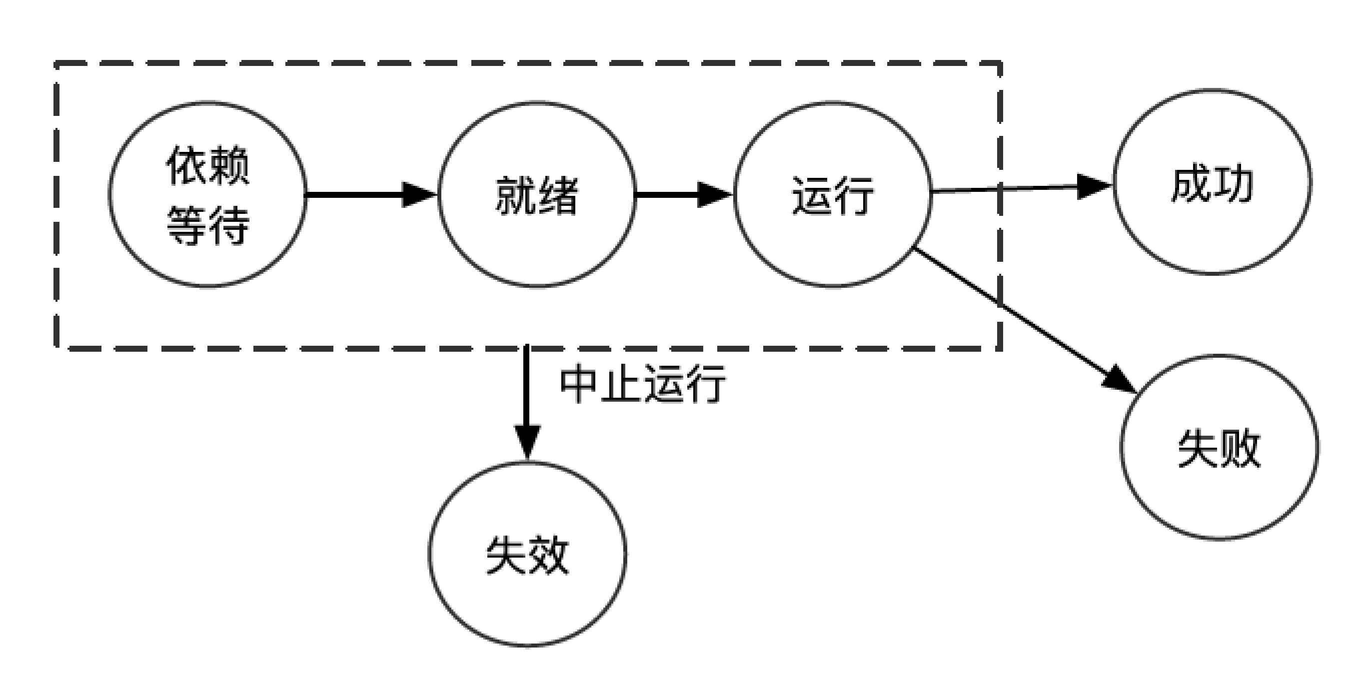


图 8-19　ETL 任务的生命周期状态图

- 依赖等待：ETL 任务刚刚生成，需要等待这个任务的上游依赖就绪。
- 就绪：ETL 任务的上游都已经就绪，等待运行资源。
- 运行：运行资源就绪，ETL 任务已经开始跑起来了。
- 成功：ETL 任务运行完毕，正常结束，没有出错。
- 失败：ELT 任务运行中途出错。
- 失效：在运行或者等待过程中的任务，可以手工中止运行，任务变为失效状态。

可以看到，对于调度程序来说，一方面要将任务按照既定的规则交给执行器去做，另一方面，需要保持每个任务的状态，并根据任务运行状况来更新任务状态。因此，调度程序是一个守护进程(或者叫后台进程)，它的伪代码如下：

```
while (true):
    if (需要生成任务):
        生成新任务&&并标记状态为“依赖等待”
    if (有任务完成):
        根据运行结果标记任务为“完成”或者“失败”
    if (有用户手工中止要求):
        中止任务运行&&标记为“失效”
    if (有“依赖等待”任务&&上游已经“完成”):
        标记任务为“就绪”
    if (有“就绪”任务&&有空闲资源):
        把任务交给执行器去运行&&标记任务为“运行”
```

实际的生产环境中，调度的逻辑会比这里的伪代码复杂得多，但是究其本质，无外乎这几行逻辑。为了满足复杂的业务需求，以及不断膨胀的 ETL 任务列表，以及有限的资源，会在调度的基础逻辑上增加很多的策略，比如 ETL 任务的类别、按照优先级调度、运行开始时间要求、任务的资源限制等。

8.6.2 需要考虑的其他方面

1. 重复执行

对于一个可用的调度系统，一个最重要的指标是它的稳定性。除了关注系统稳定性，还需要关注任务运行的稳定性。在多任务并发、资源占用率高、网络通信较多的环境下，任务的可重复执行是一个重要特性。

一方面，ETL 任务的设计和实施需要具备可重复执行的特性。这种实施是系统自动化管理的需要，也对数据变更后手工恢复提供了便利。

另一方面，调度系统需要有任务重复执行的机制。一个设计较好的调度系统，允许配置多次的重试机制，在任务失败之后重试指定的次数。一个典型的例子就是，通过不断的重试来检查一份数据是否就绪。

2. 任务故障通报

虽然有了任务重复执行，还是不可避免任务出现各种各样的问题。

典型的两种任务故障是：任务运行失败、任务运行超时。任务运行失败是指 ETL 任务遇到了非预期的状况导致异常结束；任务运行超时，是指任务一直在运行，超过了指定的时间长度还未完成。对于任务运行超时，一般的调度系统会允许设置两种处理方案：允许其继续运行，或者系统直接将任务中止掉。其中，第一种方案一般是经过谨慎评估之后才允许使用的。

对于出现的任务故障，一般会通过多种方式来通报给任务负责人。常见的有邮件、短信、电话等。一种较好的实现方案，是既兼顾使用的方便性，又能兼顾特殊任务管理的易用性，具体的任务故障通报方案依据业务需求来实现。

相关的负责人在收到任务故障的通报后，需要根据业务要求以及既定的流程来做相关的操作和恢复工作。

3. 系统恢复

除了对 ETL 任务的故障通报，调度系统也可能会遇到网络拥塞、机器故障等问题，还有可能遇到系统升级、网络迁移等预期中的事件。调度系统本身的恢复也十分重要。这里，

我们讨论的一个前提，是每个任务的状态已经有记录并可恢复。

一种最简单的方案，是时间点机制。调度系统在文件中或者数据库中记录了最新完成了调度任务的时间点，待系统恢复后，可以直接从这个时间点之后来恢复系统状态。但这种方案过于简单，对于较复杂的机制的系统恢复难以满足。

另一种方案是快照机制。调度系统会在文件中或者数据库中记录当前系统运行的快照，待系统恢复的时候，会读取快照，并尝试恢复所有的运行时状态。使用这种方案的时候，务必使快照安排在很好的保护之后，因为在系统恢复的时候一定要有。

当然，经过很好设计的调度系统，还可以把更强大的恢复功能放入另外的可靠架构中。

8.6.3 简易调度示例

上面说过了较多的调度的设计原则，下面我们将会给出一个非常简单的调度系统的设计示例。

假设我们有 10 个任务，每个任务都是从 HDFS 上的数据做数据处理，任务的详细内容就是执行一条命令，类似：hadoopfs -cat $HDFSPATH | awk ‘$YOURCMD’，所有的任务都是每天运行一次，任务之前有依赖关系。

在这样的简单示例中，我们给出如图 8-20 所示的精简调度系统设计。

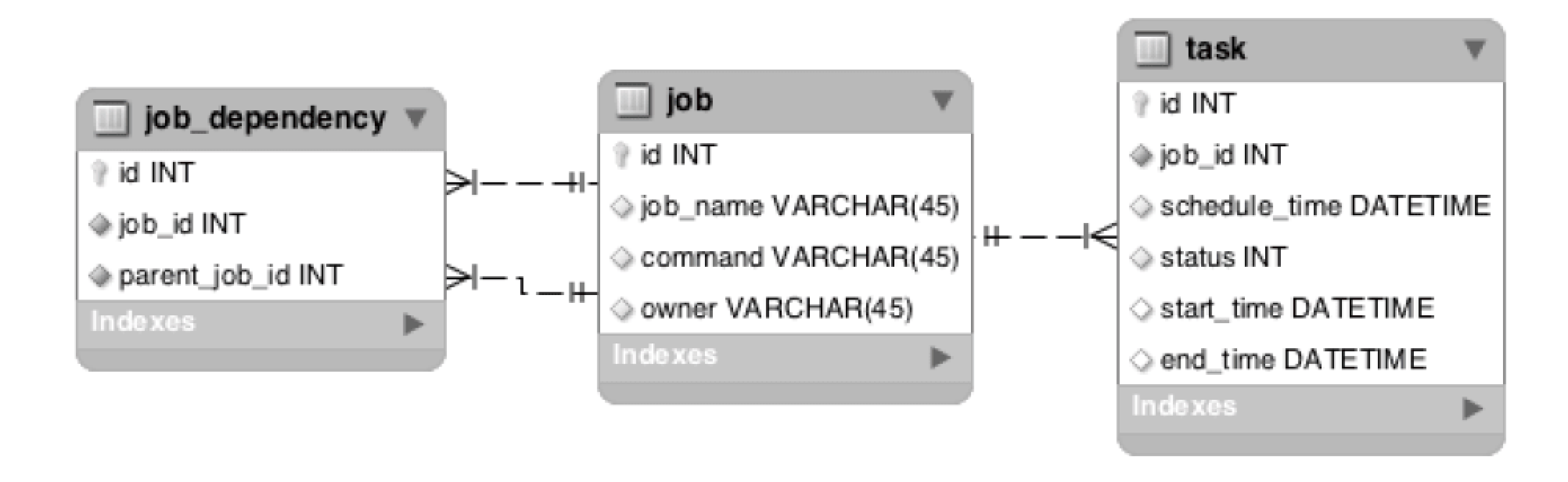


图 8-20 简洁的调度 ERD

就如同程序和进程，程序是固定的，程序的一次运行就是一个进程。在调度系统中，job 和 task 关系也是类似的，job 是一个 ETL 任务的配置，task 是这个 job 的一次运行。从图 8-20 中可以看到，每个 task 都是 job 在某一个 schedule_time 的实例。

job 之间的依赖是通过 job_dependency 来存放的，其中的 job_id 和 parent_job_id 都是外键指向 job 表的 id。

有了这三张调度信息的存储表，读者就可以根据上文中调度的伪代码来完成一个简单的调度系统了(注意，读者实现的时候可以直接把本机当作执行机)。当然，如果要真正用于生产环境的话，还需要对这个调度系统做很多改造。

8.7 数据仓库的架构

前面已经讲解了多个数据仓库按照主题来划分，讲解了数据仓库和数据集市的关系，讲解了数据仓库的三层结构，也讲解了 ETL 的概念，以及落实到工程实施后的 ETL 任务可

以在调度中运行起来。本节给出一个数据仓库整体架构的示例，如图 8-21 所示。读者可以感受一下数据处理的整体架构。

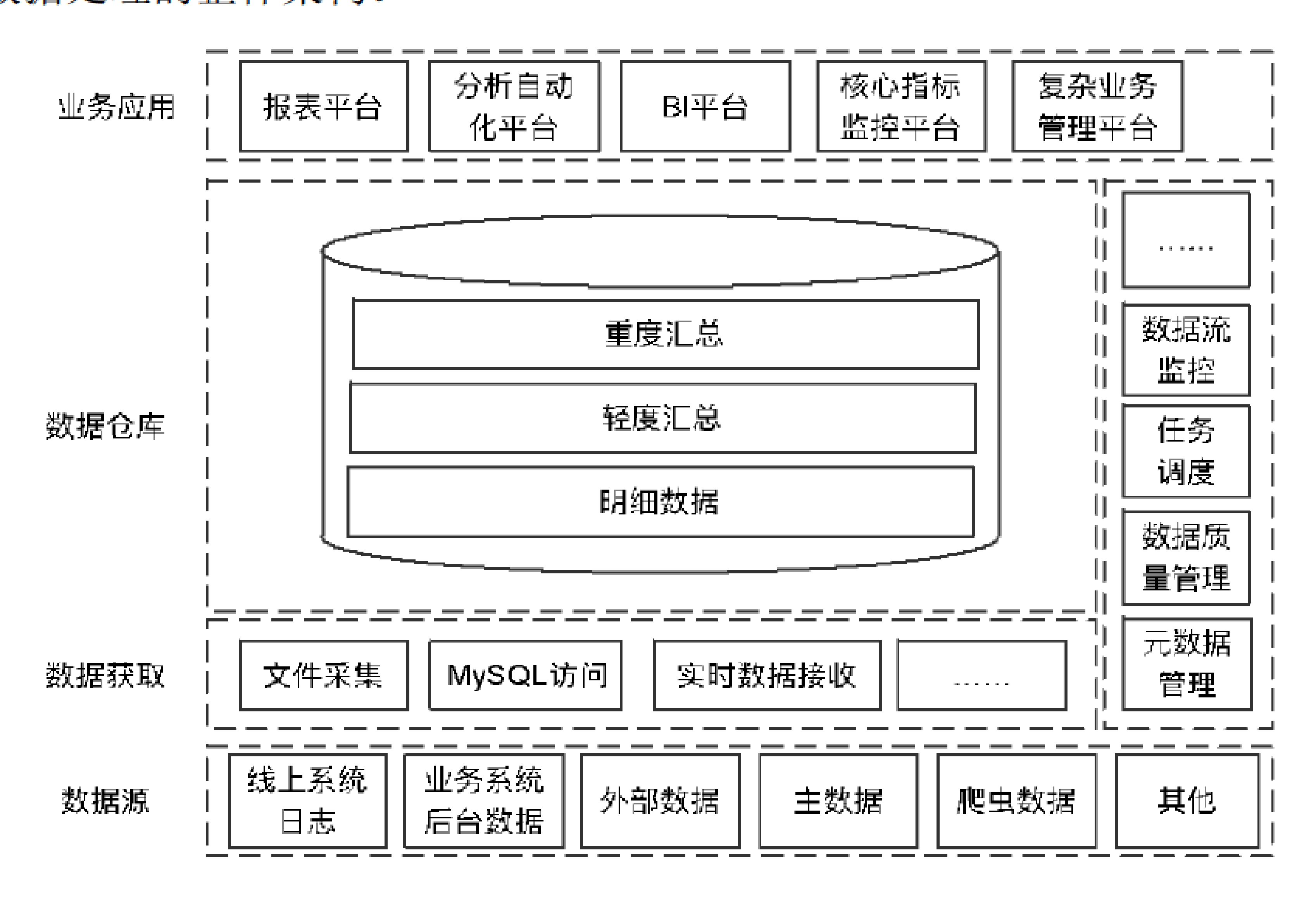


图 8-21　数据仓库架构示例

首先，可以看到架构中有四个层次，最底层的数据源是数据仓库的输入，最上层的业务应用是数据仓库的输出。其中，数据源包括线上系统的日志、业务系统的后台数据、外部数据、主数据、爬虫数据等。上层的业务应用比较广泛，例如报表平台、分析自动化平台、BI 平台、核心指标监控平台、复杂业务管理平台等。从输入和输出我们可以看出，数据仓库对接的数据源是多种多样的系统和数据内容，数据仓库上提供支持的应用更是有多种多样的类型，基本上，需要用到大量数据的地方都会使用数据仓库的数据。

中间两个层次是数据获取层和数据仓库层，以及右侧的仓库相关的公共工具，这三个部分是一个有机的整体，缺一不可。从广义上来讲，这三个部分都属于数据仓库的范畴，如果单从数据仓库呈现给用户的数据内容上来看，就是我们标注了的狭义的数据仓库的部分。狭义的数据仓库就是我们前面章节讲过的三个层次。其中，明细数据是通过多种多样的数据获取工具从数据源获取到的原始数据。而右侧的工具是对数据仓库的元数据、数据质量、数据流和调度相关工作的管理，是贯穿整个数据仓库建设始终的。

8.8　数据仓库的展望

8.8.1　数据仓库发展的阶段性

1. 用脚本看数据

在组织业务发展的初期，或者新业务刚刚发展的时候，系统数据量非常小，数据周期性变化也很小，通过单台机器足以满足看数据的需求。这个时候，通常对于有限的几类看数据需求，通过运行脚本来看数据就可以满足需求。

常见的使用方式是在 Crontab 中配置一个定时运行任务(如 Shell、Python、Perl 等)来提

取需要看的数据，通常会伴随着以邮件发送数据结果。

2. 从系统看数据

随着业务的发展，通过脚本看数据的方式已经不能满足业务上的需求，表现为：数据量膨胀很快，单台机器无法处理大量的数据；业务需求变多，脚本的方式会带来相互引用、数据交叉等管理复杂的问题。

这时，一般需要有具备一定抽象程度的数据平台。这个数据平台需要具备一定的可扩展性，能方便地完成处理数据需求的工作，需要具有脚本处理的易用性和批量管理的规范性，一般会根据业务情况的不同给予不同的方案。

这个阶段中，相比于用脚本看数据，会极大地缩短需求开发的时间，极大地方便了用户查看数据的过程。但是，这个阶段对于数据内容是疏于管理的，人们只会关注统计结果的数字。

同时，这个阶段的数据源不会过于复杂，数据仓库上面的业务应用也会相对比较简单，大部分都是具有报表展示和某种获取数据的方式就够了。

3. 数据集市建设

在进入从系统看数据的阶段中后期，一般会频繁地发生一些数据内容上的事故，例如数据统计错误、数据口径不统一导致汇总结果不一致、数据丢失等。这个时候，就迫切需要从数据内容的角度来做一个简洁的数据集市了。

通过对数据来源、数据内容等的梳理，以及相关工作的规范化，形成一个组织内认可的，有专人维护的数据集市，任何想要取得可信数据的人，都可以自助或者通过接口人，来从数据集市获取理想的数据。

这个时候，数据仓库上面的业务应用也开始慢慢变得丰富起来，除了基础的报表和取数，可能会有更贴近业务发展水平的工具集和分析示例展现出来。

4. 数据仓库建设

在数据集市刚刚建立的初期，极大地方便了组织内的数据工作，但是慢慢地，又会出现新的问题，由于没有经过规范的数据仓库建设，也没有仓库建设相关的工具支持，对数据质量的控制不够等。此时的表现，一般是数据集市团队的人员疲于应付，已经无法承担起大量的数据需求，数据的增长速度和管理水平也不匹配，导致各项工作处在一片混乱中。

这时，更强大、更规范的数据仓库建设就势在必行了。一方面，需要从头梳理数据的来龙去脉，另一方面，也需要搭建元数据管理、数据质量流程、ETL 开发工具和规范等一系列的工具和平台。同时，把组织内的数据作为数据资产，予以保护和管理。很多组织的数据部门也正是在这个时候开始大发展的。

等这一整套工具平台、流程规范都已经就绪后，相应的业务需求也会如雨后春笋般地涌现出来，各种业务形态也迫切地需要相关的数据支持。如果组织内的主流业务线比较单一，或者组织的规模没有迅速扩张，单一数据仓库的状态是基本够用的。

5. 大型数据仓库集

随着组织规模的壮大，或者业务逻辑的复杂化和业务线的拆分，大型组织内的唯一的

一套数据仓库架构和支撑体系就没有办法满足飞速增长的业务需求了。

这个时候，除了一套数据仓库架构外，对于每条业务线，还需要个性化的数据仓库/集市的支持。如果业务线划分越来越分散，可能会出现多中心数据仓库的情况，这就形成了数据仓库集。当然，更大规模的异地组织，或者没有交互的业务线之间，很可能会形成毫无关联的数据仓库集。

这个阶段，数据仓库上层的数据产品会异常丰富，除了传统大数据架构系列的数据产品、BI 产品外，很多业务线的业务产品都会出现数据仓库的数据供应的痕迹。做得更好的，设置能将数据转化为生产力，对内驱动思想，对外变现。

上面几个阶段，是数据仓库发展一般都要经历的中间形态，也有一些组织会由于业务发展的急缓和特点，而跳过其中一个或者几个阶段。但是，综合来说，需要具有极强业务理解能力和数据仓库架构能力的架构师，来把握数据介入业务的深度和范畴，从而既不会因为数据发展缓慢拖累了业务的进度，又不会因为数据发展超前而压垮了业务(大量的数据仓库及周边设施投入的开销会非常大，过早投入会占用组织内过多的资源，可能会导致复杂的管理问题等)。

8.8.2 未来的数据仓库

数据仓库是最近 5~10 年兴起的一个新领域，即使追溯相关理论，也不超过 20 年。但是，各大传统行业的数据服务提供商，以及各大互联网公司，都在数据方向耗费了巨资，来做研究，和进行生产投入。

虽然目前还有非常多的组织去自主搭建数据仓库相关的基础设施，但我们惊喜地发现，越来越多的大公司和创业者，都在想方设法将数据仓库相关的工作变得非常容易和轻量级。

相信用不了几年的时间，一方面，高校和社会机构会将大数据的教育作为一类常见的工作开展下去，稳定、高质量地向社会输送数据仓库人才；另一方面，数据仓库的基础设施会变得像我们打开操作系统一样，变得易于掌控和使用。

然而，对于数据仓库基础知识和架构的理解，仍旧是时代对高精尖技术人才的要求。

8.9 小　　结

本章首先讲解了数据仓库的由来和特性，以及数据仓库相关的概念，之后，带领读者熟悉了 ETL 和调度相关的工作，最后，通过梳理，给出了一个常见的数据仓库架构图，并提供了组织内数据仓库发展的阶段性，并设想了未来数据仓库基础设施建设的愿景。

通过本章的学习，希望读者对数据仓库能有最基本的认识，并能说清数据仓库相关概念的含义和区别，能够理解三层架构，并能基于本章给出的一些小例子，自己搭建一套简单的自下而上的数据仓库系统。

第 9 章

Hive

学习目标

本章带领读者一起认识 Hive，首先了解 Hive 是什么，如何安装和配置，然后介绍常见的 Hive 命令行接口以及常见的 Hive 操作；然后会介绍 Hive 的自定义函数以及 Hive 的一些高级使用，最后会给出 Hive 的编程示例。

通过本章的学习，读者应能够了解分布式数据仓库 Hive，能够掌握其基本配置和使用。

本章要点

- 什么是 Hive，及其用途
- Linux 下 Hive 的搭建
- Hive 下编写自定义函数
- 了解 Hive 视图、索引以及常见的优化
- Hive 编程接口

9.1 初识 Hive

分布式文件系统是支撑海量数据存储和运算的基础架构，那么，支撑数据仓库的基础架构是什么样的呢？

Hive 是各大企业中最常见的基础架构，本节带领读者一起来认识一下 Hive 的概貌。

9.1.1 Hive 是什么

Hive 也是 Apache 基金会下面的开源框架，是基于 Hadoop 的数据仓库工具，它可以把结构化的数据文件映射为一张数据仓库表，并提供简单的 SQL(Structured Query Language)查询功能，后台将 SQL 语句转换为 MapReduce 任务来运行。

Hive 和 HBase 是两种基于 Hadoop 的不同技术——Hive 是一种类 SQL 的引擎，并且运行 MapReduce 任务，HBase 是一种在 Hadoop 之上的 NoSQL 的 Key/Value 数据库。

Hive 在英文中的意思是蜂箱、蜂巢、储备、积累的意思，正与数据仓库容纳海量有价值的数据的含义相仿。图 9-1 是 Hive 的形象化标志，看起来像一只飞起的蜜蜂，但有大象的头，表明与 Hadoop 是一家，是基于 Hadoop 的数据仓库。



图 9-1 Hive 的标志

对于用惯了传统数据库(比如 MySQL、Oracle)的人来说，由于对传统的 SQL 非常了解，在开始使用 HiveSQL 的时候，上手会比较快，因为它们与基础的 SQL 结构是类似的；当然，HiveSQL 为分布式计算做了一些优化，后面的章节会详细地带领读者去实践。这也正是我们理解 Hadoop 原理、理解 Hive 原理的目的。只有理解了这些东西，我们才能很好地把它们用起来。

由于 Hive 查询使用的是 Hadoop 的 MapReduce 作业，所以大部分查询过程都是一个批量操作的过程，因此与 MapReduce 一样，会是高延迟的，所以 Hive 不适合那些低延迟的应用。同时，Hive 也不支持对已有数据的修改和追加。

9.1.2 Hive 的部署

为了让读者能更快地对 Hive 有直观的认识，我们现在来部署一套 Hive。

Hive 的官网位置是：http://hive.apache.org/，读者可以从里面的 Downloads 部分找到 Hive 的一个发布的版本。本章以 Hive 1.2.1 为例，讲解 Hive 的配置，以完成后续的实践。

1. 前期准备

确保你要使用的 Hive 机器上已经部署好了 Hadoop 客户端，并配置好了环境变量，可以使用如下命令来检查：

```
[work@localhost usr]$ hadoop version
Hadoop 2.6.0
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r
e3496499ecb8d220fba99dc5ed4c99c8f9e33bb1
Compiled by jenkins on 2014-11-13T21:10Z
Compiled with protoc 2.5.0
From source with checksum 18e43357c8f927c0695f1e9522859d6a
This command was run using
/usr/hadoop-2.6.0/share/hadoop/common/hadoop-common-2.6.0.jar
```

2. 下载文件

访问 Hive 官网，选择下载 1.2.1 版本的 Hive，我们这里选择 apache-hive-1.2.1-bin.tar.gz，如果选择 src 的压缩包，则下载的是源码，需要自己编译。将下载好的文件放入你想要使用 Hive 的机器的/usr 目录下(或者直接在 Linux 环境下下载这个压缩包)：

```
[work@localhost usr]$ cd /usr
[work@localhost usr]$ tar -zxf apache-hive-1.2.1-bin.tar.gz
```

3. 修改环境变量

编辑用户主目录下面的.bashrc 隐藏文件，在 export HADOOP_HOME 的后面增加 export PATH 的行。保存之后退出，然后执行 source：

(1) 编辑用户主目录下面的.bashrc 隐藏文件：

```
[work@localhost usr]$ vi ~/.bashrc
```

(2) 在 export HADOOP_HOME 后面增加及修改：

```
export HIVE_HOME=/usr/apache-hive-1.2.1-bin
export PATH=$JAVA_HOME/bin:$HADOOP_HOME/bin:$HIVE_HOME/bin:$PATH
[work@localhost hadoop-2.6.0]$ source ~/.bashrc
```

这个命令只需要执行一次，以后每次登录系统之后，后台会自动执行。

4. 修改配置文件

执行下列命令：

```
[work@localhost usr]$ cd /usr/apache-hive-1.2.1-bin/conf/
[work@localhost usr]$ mv  hive-env.sh.template  hive-env.sh
[work@localhost usr]$ mv hive-default.xml.template  hive-site.xml
[work@localhost usr]$ vi /usr/apache-hive-1.2.1-bin/bin/hive-config.sh
```

在文件最后增加如下三行：

```
export JAVA_HOME=/usr/jdk1.7.0_45
export HIVE_HOME=/usr/hadoop-2.6.0
export HADOOP_HOME=/usr/apache-hive-1.2.1-bin
```

保存后退出。

5. 启动 Hive

注意，在启动 Hive 之前，需要确保 Hadoop 是正常运行的：

```
[work@localhost usr]$ cd /usr/apache-hive-1.2.1-bin/
[work@localhost usr]$ cd /usr/bin/hive
Logging initialized using configuration in
jar:file:/usr/apache-hive-1.2.1-bin/lib/hive-common-1.2.1.jar!/
hive-log4j.properties
hive> show databases;
OK
default
Time taken: 1.42 seconds, Fetched: 1 row(s)
hive> create table mytest(id int, name string);
OK
Time taken: 0.739 seconds
hive> desc test;
OK
id                    int
name                  string
Time taken: 0.574 seconds, Fetched: 2 row(s)
hive> exit;
[work@localhost usr]$ hadoop fs -ls /user/hive/warehouse/
Found 1 items
drwxr-xr-x   - work supergroup          0 2016-02-24 21:26
/user/hive/warehouse/mytest
```

可以看到，我们在 Hive 里面使用了默认的 default 库，新建了一张 mytest 的表，在 HDFS 上已经可以看到这个表的目录了。

9.1.3 以 MySQL 作为 Hive 的元数据库

元数据库是存储 Hive 里面数据元信息的地方，默认情况下，Hive 会使用内嵌的 Derby 数据库作为存储引擎。Derby 引擎一次只能打开一个会话，如果切换为使用 MySQL 作为外置存储引擎，可以支持多个用户同时访问。

1. 创建 MySQL 中的 Hive 库

操作如下：

```
[work@localhost usr]$ mysql -hlocalhost -uroot -p
mysql> create database hive;
mysql> GRANT all ON hive.* TO root@'%' IDENTIFIED BY 'admin';
mysql> flush privileges;
```

2. 修改 Hive 的配置

这里从 https://dev.mysql.com/downloads/connector/j/下载 mysql-connector-java 的压缩包，并解压出 JAR 包，放在/usr/apache-hive-1.2.1-bin/lib 下面。之后修改配置文件：

```
[work@localhost usr]$ vi /usr/apache-hive-1.2.1-bin/conf/hive-site.xml
<property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>
        jdbc:mysql://localhost:3306/hive?createDatabaseIfNotExist=true
    </value>
</property>
```

```
<property>
   <name>javax.jdo.option.ConnectionDriverName</name>
   <value>com.mysql.jdbc.Driver</value>
</property>
<property>
   <name>javax.jdo.option.ConnectionUserName</name>
   <value>root</value>
</property>
<property>
   <name>javax.jdo.option.ConnectionPassword</name>
   <value>admin</value>
</property>
```

上面的配置是假设使用 root 访问 localhost 的 MySQL，如果你的 MySQL 访问串不同，可修改上面配置中的内容。

修改后，仍旧按照上面所述的方法访问 Hive，Hive 会在配置文件中指定的 MySQL 串对应的库中建立所有元数据的表。比如按上面所述，如果我们使用 root 访问 localhost 的 MySQL 库，就可以看到如下这些表：

```
mysql> show tables;
+---------------------------+
| Tables_in_hive_metastore |
+---------------------------+
| BUCKETING_COLS            |
| CDS                       |
| COLUMNS_V2                |
| COMPACTION_QUEUE          |
| COMPLETED_TXN_COMPONENTS |
| DATABASE_PARAMS           |
| DBS                       |
| DB_PRIVS                  |
| DELEGATION_TOKENS         |
| FUNCS                     |
| FUNC_RU                   |
| GLOBAL_PRIVS              |
| HIVE_LOCKS                |
| IDXS                      |
| INDEX_PARAMS              |
| MASTER_KEYS               |
| NEXT_COMPACTION_QUEUE_ID |
| NEXT_LOCK_ID              |
| NEXT_TXN_ID               |
| NUCLEUS_TABLES            |
| PARTITIONS                |
| PARTITION_EVENTS          |
| PARTITION_KEYS            |
| PARTITION_KEY_VALS        |
| PARTITION_PARAMS          |
| PART_COL_PRIVS            |
| PART_COL_STATS            |
| PART_PRIVS                |
| ROLES                     |
| ROLE_MAP                  |
| SDS                       |
| SD_PARAMS                 |
| SEQUENCE_TABLE            |
| SERDES                    |
| SERDE_PARAMS              |
| SKEWED_COL_NAMES          |
```

```
| SKEWED_COL_VALUE_LOC_MAP |
| SKEWED_STRING_LIST       |
| SKEWED_STRING_LIST_VALUES|
| SKEWED_VALUES            |
| SORT_COLS                |
| TABLE_PARAMS             |
| TAB_COL_STATS            |
| TBLS                     |
| TBL_COL_PRIVS            |
| TBL_PRIVS                |
| TXNS                     |
| TXN_COMPONENTS           |
| TYPES                    |
| TYPE_FIELDS              |
| VERSION                  |
+--------------------------+
51 rows in set (0.00 sec)
```

可以看到，Hive 默认初始化了元数据库。其中比较常用的有如下这些表。

(1) VERSION：记录 Hive 的版本号。可以尝试执行 select * from VERSION。

(2) DBS：记录 Hive 里面有哪些仓库。可以尝试执行 select * from DBS。

(3) TBLS：记录 Hive 中的所有表。可以尝试执行 select * from TBLS WHERE DBS=1。

(4) SDS：查看表所对应的 HDFS 目录的元数据。可以尝试执行 select * from SDS。

(5) PARTITIONS：查看某个表的分区信息。可以尝试执行 select * from PARTITIONS where TBL_ID=1。

(6) COLUMNS_V2：查看某表的某列信息。可尝试执行 select * from COLUMNS_V2。

(7) PARTITION_KEYS：查看某个表的 PARTITION 的列。可尝试执行 select * from PARTITION_KEYS。

小提示

大部分时候，不需要修改 Hive 的元数据库。尤其在使用 Hive 初期，一定要慎重操作元数据库。只有在 HDFS 底层变动时(如集群名称修改、某些分区无法恢复等极特殊情况)，才可能会直接修改元数据库。

9.1.4 Hive 的体系结构

Hive 是基于 Hadoop 的数据仓库框架，Hive 的数据存储在 HDFS 中，大部分的查询、计算由 MapReduce 完成(包含*的查询，比如 select * from tbl 不会生成 MapRedcue 任务)。

Hive 的用户接口主要有三个：命令行，JDBC 接口和 Web 界面。

命令行是最常用的一种访问方式。客户端启动命令行的时候，会连接 Hive 服务端所在的节点，这个节点会启动一个单独的驱动实例，驱动的实例会将用户的命令提交执行。后面会详细介绍命令行的使用。

Hive JDBC 接口提供了 JDBC 驱动给 Java 代码，Java 可以使用这个接口来进行一些类似关系型数据的 SQL 语句查询等操作。与关系型数据库一样，这个接口需要 Hive 启动一个叫作 Thrift 的服务。

Hive 也提供了 Web 界面，用户可以通过浏览器访问 Hive，做一些基础的操作。

Hive 将元数据存储在数据库中，如上面所说的 Derby 或者 MySQL。Hive 中的元数据包括表的名字、表的列和分区及其属性、表的属性(是否为外部表等)、表的数据所在目录等。

Hive 的驱动完成 HiveSQL 语句从词法分析、语法分析、编译、优化以及查询计划的生成的过程。生成的查询计划存储在 HDFS 中，并在随后由 MapReduce 调用和执行。

Hive 的体系结构如图 9-2 所示。

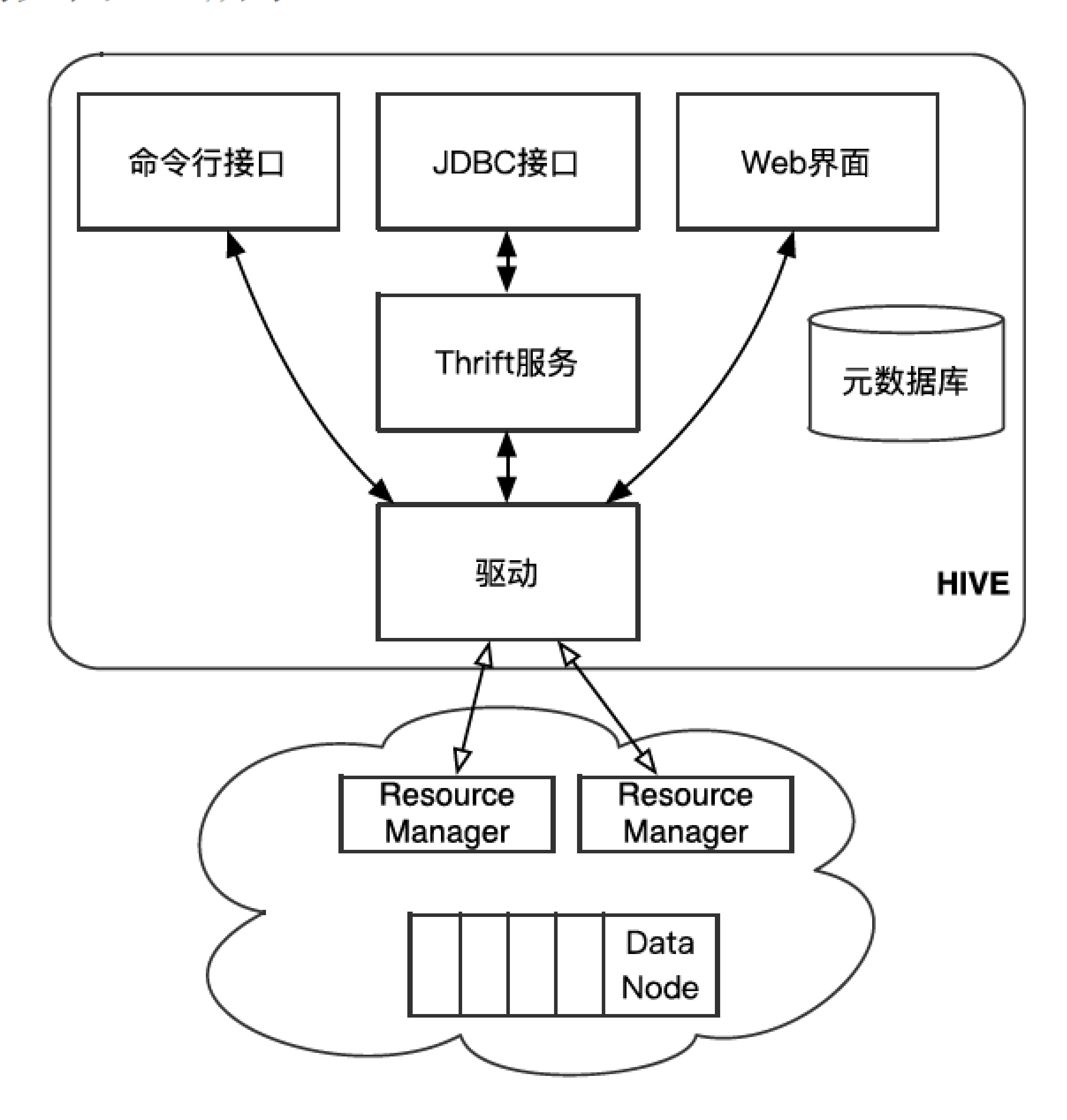


图 9-2　Hive 的体系结构

9.1.5　Web 界面展示

启用这个 Web 界面服务的过程如下。

(1)　下载源码。Hive 1.12.1 的程序包没有附带 HWI 的 War 包，我们需要到 Hive 的官网上，下载 Hive 源码文件 apache-hive-1.2.1-src.tar.gz。

(2)　打 War 包。解压后，将 hwi/web 目录下的文件用 jar cvf hive-hwi-1.2.1.war ./* 命令打包成一个 War 包。

(3)　放入 lib 目录。把 hive-hwi-1.2.1.war 放到${HIVE_HOME}/lib 目录下。

(4)　修改配置。修改${HIVE_HOME}/conf/hive-site.xml 文件。

(5)　启动服务。启动 hwi 的方式是：${HIVE_HOME}/bin/hive --service hwi。

(6)　浏览器访问。启动之后，就可以在浏览器访问“http://localhost:9999”了。

访问界面如图 9-3 所示，单击其中的 Browse Schema，可以看到 Database 列表，只有 default 一个库。

单击其中的 default，可以看到 Table 列表，只有 mytest(我们前面建立的表)。

单击 mytest，就可以看到这张表的详细描述。

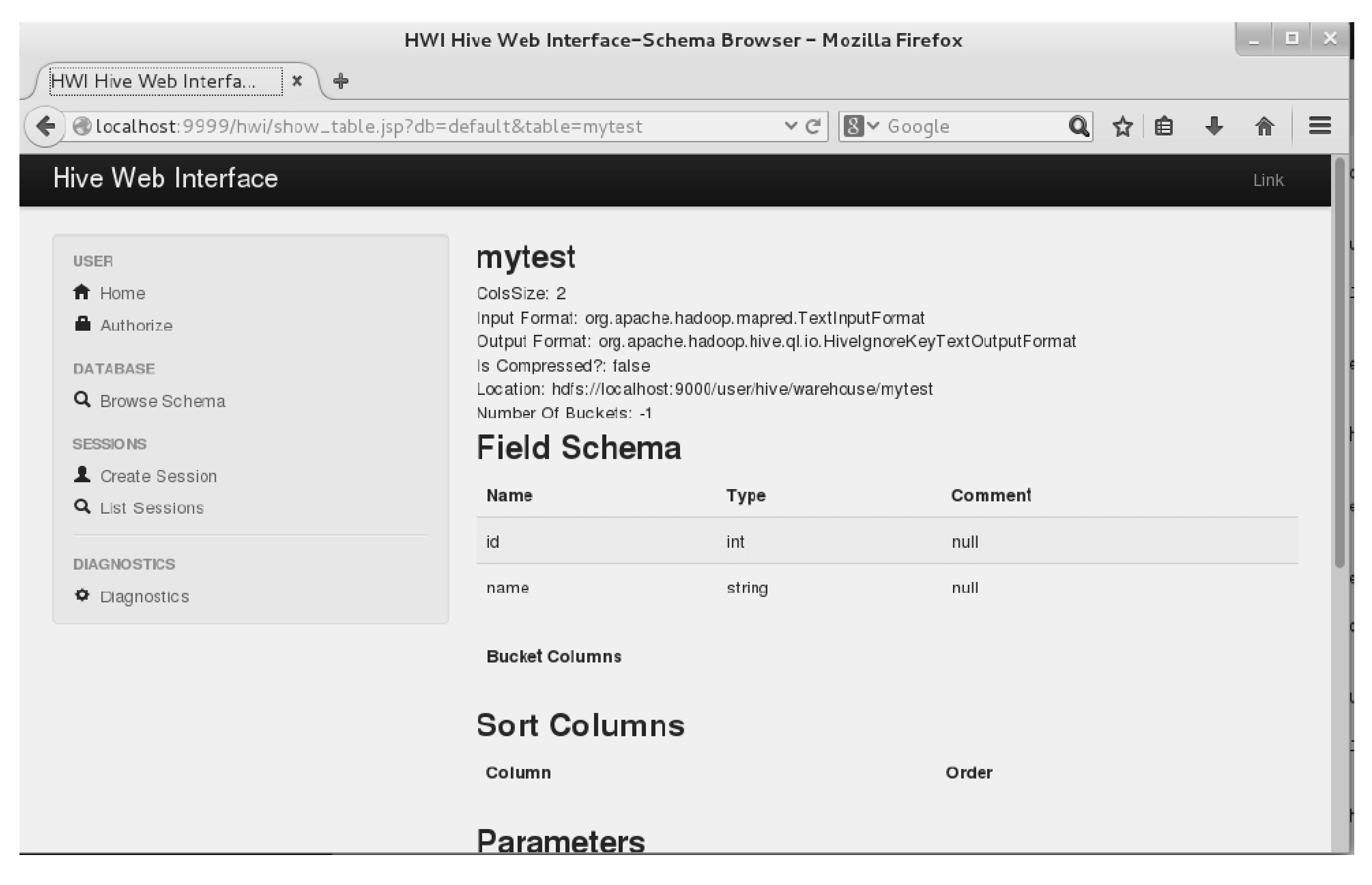


图 9-3　Hive 的 Web 界面访问

9.2　Hive 命令行接口

9.2.1　启动 Hive 命令行

命令行接口的启动方式如下：$HIVE_HOME/bin/hive 或者 $HIVE_HOME/bin/hive --service cli。可以通过输入$HIVE_HOME/bin/hive --H 来获取可用参数的帮助：

```
[work@localhost usr]$ hive -H
usage: hive
 -d,--define <key=value>        Variable substitution to apply to hive
                                 commands. e.g. -d A=B or --define A=B
    --database <databasename>    Specify the database to use
 -e <quoted-query-string>        SQL from command line
 -f <filename>                   SQL from files
 -H,--help                       Print help information
    --hiveconf <property=value> Use value for given property
    --hivevar <key=value>        Variable substitution to apply to hive
                                 commands. e.g. --hivevar A=B
 -i <filename>                    Initialization SQL file
 -S,--silent                     Silent mode in interactive shell
 -v,--verbose                    Verbose mode (echo executed SQL to the
                                  console)
```

这些命令参数的具体含义如下。

(1)　-d,--define：定义一个可以在 Hive 命令行中使用的变量。

(2)　--database：进入 Hive 命令行的时候指定数据库，如不指定，会进入 default。

(3)　-e：执行一个 SQL 语句。

(4) -f：执行 filename 文件中的 HiveSQL 语句。

(5) -H, --help：显示帮助信息。

(6) -h：连接 hostname 指定的远程 Hive 服务。

(7) --hiveconf：设置 Hive 命令行的运行时配置参数，优先级高于 hive-site.xml，但低于 Hive 命令行中使用 set 命令的设置。

(8) --hivevar：同--define。

(9) -i：进入 Hive 命令行时，先执行 filename 中的 HiveSQL 语句。

(10) -S, --silent：静默模式，不显示执行进度，最后只显示结果。

(11) -v, --verbose：冗余模式，额外打印出执行的 HiveSQL 语句。

9.2.2 可用的命令

在 Hive 命令行模式中，可以使用如下这些命令。

(1) quit、exit：退出命令行模式。

(2) reset：重置所有 Hive 运行时配置参数，比如，先前使用了 set 设置的值，会恢复成 hive-site.xml 中指定的配置值。

(3) set <key>=<value>：设置 Hive 运行时配置参数。

(4) set, set -v：输出当前的 Hive 参数设置。

(5) add FILE[S]/JAR[S]/ARCHIVE[S] <filepath>+：向 Hive 的分布式缓存中添加一个或者多个文件、JAR 包或者归档，添加后，可以在后续使用。比如，自定义一个 UDF 函数，打成 JAR 包，在创建函数前，需要使用 add jar 的方式，先将 JAR 包添加到分布式缓存。

(6) list FILE[S]/JAR[S]/ARCHIVE[S]：列出分布式缓存中的资源。

(7) list FILE[S]/JAR[S]/ARCHIVE[S] <filepath>+：检查分布式缓存中是否存在资源。

(8) delete FILE[S]/JAR[S]/ARCHIVE[S]：从分布式缓存中删除指定的资源。

(9) !<command>：在命令行模式中执行一个 Linux Shell 命令。

(10) dfs <command>：在命令行模式中执行一个 Hadoop dfs 命令。

(11) query：执行一个 HiveSQL。

(12) source FILE <filepath>：在命令行模式中执行 filepath 指定的 Hive 脚本文件。

9.3 Hive 数据类型与常见的结构

9.3.1 数据类型

1. 基本数据类型

Hive 中的基本数据类型如表 9-1 所示。

表 9-1 Hive 的基本数据类型

类 型	所占字节	数值范围及其他说明
TINYINT	1 字节有符号整数	−128~127
SMALLINT	2 字节有符号整数	−32768~32767

续表

类　型	所占字节	数值范围及其他说明
INT	4 字节有符号整数	−2147483648 ~ 2147483647
BIGINT	8 字节有符号整数	−9223372036854775808 ~ 9223372036854775807
FLOAT	4 字节单精度浮点数	
DOUBLE	8 字节双精度浮点数	
DECIMAL	V0.11.0 引入了 38 位精度。 V0.13.0 引入了自定义精度并可扩展	默认是 DECIMAL(10, 0)，可以自定义，如 DECIMAL(9, 7)
TIMESTAMP		Hive 0.8.0 引入
DATE		Hive 0.12.0 引入
STRING		
VARCHAR		Hive 0.12.0 引入
CHAR		Hive 0.13.0 引入
BOOLEAN		
BINARY		Hive 0.8.0 引入

2. 基本类型转换

Hive 支持基本类型的转换，低字节基本类型可转化为高字节的类型，例如 TINYINT、SMALLINT、INT 可以转化为 FLOAT。而所有的整数类型、FLOAT 以及 STRING 类型可以转化为 DOUBLE 类型。

这些转化可以从 Java 语言的类型转化考虑，因为 Hive 就是用 Java 编写的。当然，也支持高字节类型转化为低字节类型，这就需要使用 Hive 的自定义函数 CAST 了。

3. 复杂类型

Hive 的复杂数据类型如表 9-2 所示。

图 9-2　Hive 的复杂数据类型

类　型	说　明	示例和访问方式
ARRAY	有序的类型相同的元素集合。 如：Array<INT>	Array(1, 2)、Array(“abc”, “d”) 访问方式 Array[n]，下标 n 从 0 开始
MAP	无序的键值对。键的类型必须相同，值的类型也必须相同。 如：Map<STRING, INT>	Map(“abc”, 1, “d”, 4)。其中，逗号分隔的元素分别为 key1、value1、key2、value2。 访问方式为 M[key]，key 是 Map 中的 key 值
STRUCT	有名称的元素集合。 如：STRUCT<name:STRING, age:INT>	Struct(“John”, 31) 访问方式为 S.item，item 是 Struct 中的元素

9.3.2　文件的存储结构

Hive 常见的存储结构是 TEXTFILE。在 Hive 建表时，可以通过 Stored As FILE_FOMAT 来指定文件存储结构。比如 Stored As TextFile。

1. TEXTFILE

这是 Hive 的默认存储格式，文件是以纯文本的方式存放在 HDFS 中的，数据不做压缩，磁盘开销比较大，数据解析的开销也比较大。可以结合 Gzip、Bzip2 使用这种压缩模式，系统会自动检查是否是压缩文件，并在使用数据的时候自动解压。它对应的输入和输出类是 org.apache.hadoop.mapred.TextInputFormat 和 org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat。使用方法比较简单，比如我们在/home/work/aaa 文件中存储三行数据：

```
hive> CREATE TABLE test1(theStr STRING)
    > STORED AS TEXTFILE;
OK
Time taken: 0.434 seconds
hive> LOAD DATA LOCAL INPATH '/home/work/aaa' INTO TABLE test1;
Loading data to table test.test1
Table test.test1 stats: [numFiles=1, totalSize=12]
OK
Time taken: 1.294 seconds
hive> select * from test1;
OK
111
222
333
Time taken: 0.553 seconds, Fetched: 3 row(s)
```

2. SEQUENCEFILE

SEQUENCEFILE 是 Hadoop API 提供的一种二进制文件支持，它具有方便、可分割、可压缩的特点。SEQUENCEFILE 具有三种压缩选项：NONE、RECORD、BLOCK。

其中，RECORD 压缩率低，一般建议使用 BLOCK 压缩。

在 Hive 读写数据时使用的是这两个类：org.apache.hadoop.mapred.SequenceFileInputFormat 和 org.apache.hadoop.hive.ql.io.HiveSequence FileOutputFormat。使用方法如下：

```
hive> CREATE TABLE test2(str STRING)
    > STORED AS SEQUENCEFILE;
OK
Time taken: 0.574 seconds
hive> SET hive.exec.compress.output=true;
hive> SET io.seqfile.compression.type=BLOCK;
hive> INSERT OVERWRITE TABLE test2 SELECT * FROM test1;
hive> select * from test2;
OK
111
222
333
Time taken: 0.97 seconds, Fetched: 3 row(s)
```

这个时候，我们查看 HDFS 上对应的数据，已经是不直接可读的文件格式了。查看方法是：desc formatted test2，查看表对应的 HDFS 目录，之后执行 Hadoop Shell 命令查看。

SEQUENCEFILE 这种二进制文件，使用 Hadoop 标准的 Writable 接口实现序列化和反序列化。Hive 中的 SequenceFile 继承自 Hadoop API 的 SequenceFile，不过，它的 key 为空，使用 value 存放实际的值，这样是为了避免 MR 在运行 map 阶段的排序过程。

这种文件结构主要由一个 Header 后跟多条 Record 组成。其中，Header 主要包含了 Key-Value 对的类名、压缩算法、用户自定义元数据等信息。Record 以 Key-Value 的方式进行存储，一般包含 Record 长度、Key 长度、Key 值和 Value 值。如果数据有压缩，则根据压缩的情况，Value 的内容会是不同的。

3. RCFILE

RCFILE 是一种行列存储相结合的存储方式。

首先，它将数据按行分块，保证同一个 record 在一个块上，避免读一个记录时需要读取多个 block。

其次，块数据列式存储，有利于数据压缩和快速的列存取。读写数据的类是 org.apache.hadoop.hive.ql.io.RCFileInputFormat 和 org.apache.hadoop.hive.ql.io.RCFileOutputFormat。它的使用方式与 SEQUENCEFILE 类似：

```
hive> CREATE TABLE test2(str STRING)
    > STORED AS RCFILE;
hive> INSERT OVERWRITE TABLE test2 SELECT * FROM test1;
```

4. 其他

Hive 还支持 Parquest、Avro 等其他格式，这里不再详细叙述，感兴趣的读者可以搜索相关资料，或者阅读 Hive 的源码。当然，我们也可以实现 inputformat 和 outputformat 来自定义输入输出格式。

在选择文件格式的时候，一方面要考虑文件的存储空间，另一方面，要考虑读取速度，因此，要选择合适的存储格式和压缩方法。一般来说，在企业的集群里，都不会直接使用 TEXTFILE 这种浪费空间的方式来存储，或者至少是对历史数据有压缩存储的方案。当然，在选择之前，需要做详细的数据对比，要使得空间占用和任务运行效率都得到业务方和预算的认可。

9.4 HiveSQL

9.4.1 数据定义语言 DDL

1. 分库、分表和分区

我们知道，在传统的数据库(如 MySQL)中，是有 Database 概念的，类似于编程语言中的命名空间。它的作用是将关联紧密的表放在一个区域之内，便于管理。比如，属于销售类的表放入 sales 这个 Database 里面，属于用户相关的表放入 users 这个 Database 里面。而且可以动态地增加 Database。

在 Hive 中，也一样有 Database 的概念，使用方式如下：

```
--创建名称为 databasename 的库
```

```
CREATE DATABASE databasename;
--使用名称为 databasename 的库
USE DATABASE databasename;
--删除名称为 databasename 的库
DROP DATABASE databasename;
```

与 MySQL 类似，在 Database 里面可以建表，每张表代表不同含义的数据集合，每张表里面有多个字段，每个字段对应上面章节中讲述的数据类型。

在查询一张 Hive 表的时候，通常会扫描一整张表所有的内容，会消耗很多时间做没必要的工作。而大部分时候，我们只关心一张表中的一部分数据，这个时候，就可以使用分区。使用分区字段将一整张表的数据分隔开来，每个分区字段可以有一个或者多个分区值，Hive 实际在 HDFS 上存储的时候，会将数据按照分区值的不同，存储在对应的目录下。同时，我们对 Hive 表中数据的操作，如装载和删除，可以以一个最小分区为单位来做。

注意，表名和分区名不区分大小写。

2. 建表

下面是个建表语句的例子，顾客订单表有 5 个字段：

```
CREATE TABLE customOrder(
    type INT COMMENT "订单类别",
    saleManId BIGINT COMMENT "售货员 id",
    customerId BIGINT COMMENT "顾客 id",
    goodId BIGINT COMMENT "商品 id",
    price DECIMAL(10,2) COMMENT "价格"
)
PARTITIONED BY (`p_date` string COMMENT "日期")
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n';
```

上面的建表语句是最经常使用的一种建表方式，没有指定 HDFS 上文件存储位置的内部表(请注意，内部表的文件存储位置也可以指定)，该表的存储位置是默认的 Hive 库地址+表地址，可以使用如下命令查看这张表的详情：

```
hive>desc formatted customOrder;
OK
# col_name             data_type              comment

type                   int                    订单类别
salemanid              bigint                 售货员 id
customerid             bigint                 顾客 id
goodid                 bigint                 商品 id
price                  decimal(10,2)          价格

# Partition Information
# col_name             data_type              comment

p_date                 string

# Detailed Table Information
Database:              tmp
Owner:                 work
CreateTime:            Mon Mar 21 15:37:20 CST 2016
LastAccessTime:        UNKNOWN
Protect Mode:          None
```

```
Retention:          0
Location:
hdfs://hlg-muce3/user/hive/warehouse/tmp.db/customorder
Table Type:         MANAGED_TABLE
Table Parameters:
  transient_lastDdlTime  1458545840

# Storage Information
SerDe Library:       org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
InputFormat:         org.apache.hadoop.mapred.TextInputFormat
OutputFormat:
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
Compressed:          No
Num Buckets:         -1
Bucket Columns:       []
Sort Columns:         []
Storage Desc Params:
  field.delim          \t
  line.delim           \n
  serialization.format  \t
```

Hive 表有两种类型：内部表和外部表。上面可以看到 Table Type 是 MANAGED TABLE，表明这个表是个内部表。

内部表的分区数据与 HDFS 存储数据强关联，表删除或者分区删除的时候，对应的数据也会删除；外部表就是 Hive 只通过指定的位置来加载表中的数据，而表删除的时候，对应的数据不会删除。

使用外部表，需要在建表的时候指定 external 关键字，而且必须指定 location 属性：

```
CREATE EXTERNAL TABLE customOrder(
...
LOCATION '$HDFSPATH'
```

应注意，虽然这里指定了 HDFS 路径，但是，Hive 并不会将路径直接关联成为 customOrder 表的 PARTITION，而是需要通过 ADD PARTITION 操作来进行关联。

3. 修改表结构

在 Hive 里面，修改表结构的操作与传统数据库类似：

```
ALTER TABLE tableName RENAME TO newName
ALTER TABLE tableName ADD COLUMNS (colSpec[, colSpec ...])
ALTER TABLE tableName DROP [COLUMN] columnName
ALTER TABLE tableName CHANGE columnName newName newType
ALTER TABLE tableName REPLACE COLUMNS (colSpec[, colSpec ...])
```

应注意，Hive 里面增加、删除字段的操作只能是所有非分区字段中的最后一个。对于新增加的字段，Hive 对于已有数据，会认为这个字段是 NULL。

4. 删除表

Hive 中删除表的操作与传统数据库类似，执行后，内部表关联的 HDFS 上的数据会一并被删除：

```
DROP TABLE [IF EXISTS] table_name;
```

9.4.2　数据操纵语言 DML

1. 分区操作

如果需要从 HDFS 某个目录加载数据到表的分区中，可以使用如下方式。如果是从执行 Hive Shell 的机器上 load 数据的话，可以使用 LOAD DATA LOCAL INPATH '$LOCALPATH'的方式。注意：$HDFSPATH 和$LOCALPATH 可以指定一个目录，也可以指定一个文件。

```
LOAD DATA INPATH '$HDFSPATH' INTO TABLE customOrder PARTITION
(p_date='20160101');
```

如果希望将 HDFS 上的数据加载到表的一个分区上，则可以使用如下方法。使用下面这种方法，添加分区的时候，不会复制/移动数据，而是直接将 Hive 元数据库中的这个 PARTITION 指向$HDFSPATH。

```
ALTER TABLE customorder ADD PARTITION (p_date='20160101') LOCATION
'$HDFSPATH';
```

另外一种添加分区的方法，是使用查询结果来生成分区数据：

```
INSERT OVERWRITE TABLE customOrder PARTITION(p_date='20160101') SELECT ...
```

这里还有一个概念，叫动态分区，它是根据 SELECT 子句的结果来决定分区的字段值，我们需要在执行 HiveSQL 之前设定 hive.exec.dynamic.partition=true;才能使用。

Hive 添加分区的时候，不会对数据进行任何检查，只是简单地在 Hive 元数据库中增加一条记录。如果源文件格式不正确，也只有在做查询操作时候才能发现，对于无法识别的字段，会以 NULL 来显示。

要查看一张表的所有分区，可以使用 SHOW PARTITIONS customorder。对于多个分区字段的，也可以在后面添加 PARTITION(p_date='20160101')来限定只展示符合这个分区限定的所有分区。

如果希望删除分区，可以使用如下方法。注意，对于内部表来说，相应$HDFSPATH 上的数据也会被删除掉。

```
ALTER TABLE customOrder DROP PARTITION (p_date='20160102');
```

2. 查询操作

Hive 里面的 SELECT 操作，基本思路与传统数据库也是类似的，基本语法如下所示：

```
SELECT [ALL | DISTINCT] selectExpr, selectExpr, ...
FROM tableReference
[WHERE whereCondition]
[GROUP BY colList | [HAVING havingCondition]]
[CLUSTER BY colList] | [DISTRIBUTE BY colList][SORT BY colList]
[ORDER BY colList]
[LIMIT number]
```

ALL 和 DISTINCT 指定重复的行是否都返回，默认是 ALL(即所有的行都返回)。WhereCondition 是一个返回 boolean 的表达式。ORDER BY 是全局排序。SORT BY 是数据

在进入 reducer 之前排序，当然，如果只有一个 reducer 的话，就是整体有序的。

DISTRIBUTE BY 是指执行任务的时候，对数据按照它指定的列来分发到不同的 reducer 来处理。CLUSTER BY 兼具 DISTRIBUTE BY 和 SORT BY 的功能，但是，排序只能是倒序排列，不能指定排序规则。

HiveSQL 会有一些为转为分布式计算而约定的一些特殊情况。下面详细说明使用 HiveSQL 的一些注意事项。

(1) 不支持等值连接。

传统数据库中，经常使用 FROM A, B WHERE A.a1 = B.b1 这样的操作，在 HiveSQL 中，需要使用 FROM A [left | right | full outer] join B on (A.a1=B.b1)这种方式。

(2) JOIN 操作。

下面使用两张表来讲解 JOIN 的差别，如表 9-3 所示。

表 9-3　两张示例表的数据

	表 A(字段 a1、a2)	表 B(字段 b1、b2)
数据	1　a 2　a 1　b	2　n 3　c

下面是这两张表各种形式关联的结果，如表 9-4 所示。

表 9-4　两张表关联的结果

	A.a1　A.a2　B.b1　B.b2	说　明
A JOIN B ON (A.a1=B.b1)	1　m　1　n 1　n　1　n	两个表的列直接关联，只有找到相等数据的会保留
A LEFT OUTER JOIN B ON (A.a1=B.b1)	1　m　1　n 1　n　1　n 2　m　NULL　NULL	左外关联：左表的所有行都会保留，能够在右表中找到匹配的行则关联，否则右表字段为 NULL
A RIGHT OUTER JOIN B ON (A.a1=B.b1)	1　m　1　n 1　n　1　n NULL　NULL　3　c	右外关联：右表的所有行都会保留，能够在左表中找到匹配的行则关联，否则左表字段为 NULL
A FULL OUTER JOIN B ON (A.a1=B.b1)	1　m　1　n 1　n　1　n NULL　NULL　3　c 2　m　NULL　NULL	全外关联：左右表的所有行都会被保留，关联不上的字段为 NULL
A LEFT SEMI JOIN B ON (A.a1=B.b1)	1　m 1　n	左半连接，类似于传统数据库中的 IN 操作，只保留左表字段中在右表字段出现过的行。而且 SELECT 和 WHERE 子句中不能出现右表中的字段

(3) NULL。

传统数据库中，NULL 通常表示空值，在 Hive 里面，如果字符串为空(即长度为 0)，对它进行判断的时候，IS NULL 结果会是 False。

(4)　数据类型转换。

常见的数据类型转换，如前面介绍数据类型所述，如果需要显式转化，可以这样做：CAST(columnName AS double)。

(5)　不能 INSERT INTO, UPDATE, DELETE。

在 Hive 里面，只能 LOAD 或者 INSERT OVERWRITE INTO 整张表或者整个分区，而不能在现有的表或者分区中 INSERT　INTO 几行数据。也不允许执行 UPDATE 更新操作。对于删除数据，也必须是批量删除整张表或者整个分区，不能 DELETE 几条。

这里的限制，充分说明了 Hive 作为数据仓库框架通常只支持一次写多次读的操作。

3. 执行计划

一个 HiveSQL，在执行的时候，是往 Hadoop 上提交一个或多个前后关联的 MapReduce 任务，来完成整个 HiveSQL 的任务处理需求。当我们写的 HiveSQL 比较复杂的时候，想要查看 HiveSQL 在 Hadoop 上的整个执行过程(比如划分为几个阶段，分别做什么)，可以使用 explain。

例如，原始 SQL 如下：

```
select * from customOrder where p_date=20160101 and price>10;
```

如果执行这条语句，结果是通过 SQL 解析，输出原始 SQL 对应的执行计划，但是，不会真正去执行查询任务：

```
hive> explain select * from customOrder where p_date=20160101 and price>10;
OK
STAGE DEPENDENCIES:
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-0
    Fetch Operator
      limit: -1
      Processor Tree:
        TableScan
          alias: customorder
          Statistics: Num rows: 1 Data size: 122 Basic stats: PARTIAL
            Column stats: NONE
          Filter Operator
            predicate: (price > 10) (type: boolean)
            Statistics: Num rows: 1 Data size: 122 Basic stats: COMPLETE
              Column stats: NONE
            Select Operator
              expressions: type (type: int), salemanid (type: bigint),
                customerid (type: bigint), goodid (type: bigint),
                price (type: decimal(10,2)), '20160101' (type: string)
              outputColumnNames: _col0, _col1, _col2, _col3, _col4, _col5
              Statistics: Num rows: 1 Data size: 122 Basic stats: COMPLETE
                Column stats: NONE
              ListSink

Time taken: 0.194 seconds, Fetched: 20 row(s)
```

我们可以看出：这个 SQL 只有 Stage-0 这样一个阶段，处理方法是扫描表，其中的过

滤器操作是 price>10，选择器是该表的所有字段，Hive 内部会把它们称为_col0、_col1 等。

细心的读者可能会发现，过滤器中并没有限定 20160101，而选择器结果的最后一个字段却是 20160101，这是因为，Hive 的不同分区是不同的目录，当我们在 SQL 里面指定了分区条件后，则对应的任务的输入就是这个分区目录的数据，同时，选择的最后一个字段也就是这个分区值。

小提示

读者在实战过程中会发现，分区值一般会在目录名里面，在具体的数据文件中是没有分区字段的。同时，这也是为什么我们修改表结构增加的字段总是在分区字段前面的原因。

4. 查看描述

查看数据库描述的语法如下，其中第二条语句在 Hive 0.15.0 以上的版本中可以使用：

```
DESCRIBE DATABASE [EXTENDED] db_name;
DESCRIBE SCHEMA [EXTENDED] db_name;
```

例如：

```
hive>desc database default;
OK
default Default Hive database hdfs://localhost:9000/user/hive/warehouse
public  ROLE
Time taken: 0.141 seconds, Fetched: 1 row(s)
```

查看表、视图和列的语法如下(下面的语句只在 Hive 0.x.x 和 Hive 1.x.x 版本中使用)：

```
DESCRIBE [EXTENDED|FORMATTED][db_name.]table_name[.col_name([.field_name]
|[.'$elem$'] | [.'$key$'] | [.'$value$'] )*]
DESCRIBE [EXTENDED|FORMATTED][db_name.]table_name[ col_name([.field_name]
| [.'$elem$'] | [.'$key$'] | [.'$value$'] )*];
```

例如，下面是直接 describe 和查看格式化的更多详细信息对比：

```
hive>desc mytest;
OK
id                     int
name                   string
Time taken: 0.177 seconds, Fetched: 2 row(s)
hive> desc formatted mytest;
OK
# col_name              data_type            comment

id                     int
name                   string

# Detailed Table Information
Database:               default
Owner:                  work
CreateTime:             Fri Feb 26 01:24:31 PST 2016
LastAccessTime:         UNKNOWN
Protect Mode:           None
Retention:              0
Location:               hdfs://localhost:9000/user/hive/warehouse/mytest
```

```
Table Type:          MANAGED_TABLE
Table Parameters:
  COLUMN_STATS_ACCURATE  false
  numFiles              0
  numRows              -1
  rawDataSize          -1
  totalSize             0
  transient_lastDdlTime  1456478671

# Storage Information
SerDe Library:       org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
InputFormat:         org.apache.hadoop.mapred.TextInputFormat
OutputFormat:
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
Compressed:          No
Num Buckets:         -1
Bucket Columns:      []
Sort Columns:        []
Storage Desc Params:
  serialization.format  1
Time taken: 0.158 seconds, Fetched: 32 row(s)
```

关于更多的信息，如查看 partition 的信息等，读者可到如下地址查看：

```
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#Lang
uageManualDDL-DescribeDatabase
```

5. 执行参数设置

在执行 HiveSQL 优化的时候，经常需要指定各种参数，指定方法为：

```
SET 参数名=参数值;
```

下面是一些常见的参数，读者可以根据 HiveSQL 优化的需求选择性地添加：

```
set hive.exec.dynamic.partition=true; //设置允许动态分区
set hive.exec.dynamic.partition.mode=nonstrict; //如果为 strict，则必须指定一
个静态分区，nonstrict 则不做要求
set hive.exec.default.partition.name=NULL; //默认的动态分区名称，当动态分区列为
''或者 null 的时候，使用设置的这个值
set hive.exec.reducers.max=200; //最多并行的 reducer 个数
set hive.exec.parallel=true; //是否开启 Map Reduce 的并发提交，默认为 false
set hive.exec.parallel.thread.number=8; //并发线程的个数
set hive.groupby.skewindata=true; //数据倾斜时负载均衡，当选项设定为 true 时，生
成的查询计划会有两个 MRJob。第一个 MRJob 中，Map 的输出结果集合会随机分布到 Reduce 中，
每个 Reduce 做部分聚合操作，并输出结果，这样处理的结果，是相同的 GroupBy Key 有可能被
分发到不同的 Reduce 中，从而达到负载均衡的目的；第二个 MRJob 再根据预处理的数据结果，按
照 GroupBy Key 分布到 Reduce 中(这个过程可以保证相同的 GroupBy Key 被分布到同一个
Reduce 中)，最后完成最终的聚合操作
set hive.merg.mapfiles=true; //合并 map 输出
set hive.merge.mapredfiles=false; //合并 reduce 输出
set hive.merge.size.per.task=256*1024*1024; //合并文件的大小
set hive.mergejob.maponly=true; //是否启用 Map Only 的合并 job
set hive.merge.smallfiles.avgsize=16000000; //文件的平均大小小于该值时，会启动
一个 MR 任务执行 merge
set mapred.reduce.tasks=20; //reduer 的个数
set hive.exec.reducers.bytes.per.reducer=1G; //每个 reduce 任务处理的数据量
set hive.exec.reducers.max=999; //每个任务最大的 reduce 数目。reducer 数=min(本
参数，总输入数据量/上一个参数)
```

6. 运算符

(1) 关系型运算符。

关系运算符包括三个判等关系运算符=、==、<==>，两个判不等运算符<>和!=，大于小于运算符>、<、>=、<=，属于区间的运算符 BETWEEN ... AND ...，判断 NULL 的运算符 IS NULL、IS NOT NULL，匹配运算符 LIKE、RLIKE、REGEXP。它们的含义我们不详细讲解，读者在使用的时候，注意 NULL 值对于运算结果的影响，这是在 HiveSQL 开发中常见的容易出错的地方。

(2) 算数运算符。

HiveSQL 中支持+、-、*、/、%(求余)、&(按位与)、|(按位或)、^(按位异或)、~(求反)。

(3) 逻辑运算符。

HiveSQL 中支持 AND、&&(同 AND)、OR、||(同 OR)、NOT(求反)、!(也表示求反)、IN(在集合中)、NOT IN(不在集合中)、EXISTS(存在至少一行)、NOT EXISTS(不存在)。

应注意，IN、NOT IN、EXISTS、NOT EXISTS 这几个运算符是在 Hive 0.13 版本之后才能支持的。如果有使用先前版本的 Hive，可以使用其他操作来实现相关的功能。如通过 LEFT OUTER JOIN 之后，选取是否右表列为 NULL 来实现 IN 操作等。IN 操作的例子如：

```
SELECT *
FROM A
WHERE A.a IN (SELECT foo FROM B) OR A.b IN ("A", "B", "C");
```

(4) 复杂类型的运算符。

可以通过 map(key1, value1, key2, value2, ...)的方式构造一个 map，通过 struct(value1, value2, value3, ...)的方式来构造一个 STRUCT(构造出的 STRUCT 的字段名是 col1、col2、col3、col4)，通过 name_struct(name1, value1, value2, value3, ...)的方式来构造一个 STRUCT，通过 array(value1, value2)的方式构造一个数组，通过 create_union(tag, value1, value2, value3, ...)的方式来构造一个 tag，指向 value 的 union。

可以使用 Array[n]的方式获取 array 中下标为 n 的元素，可以是使用 Map[key]的方式获取 map 中键值为 key 的元素。我们看一个下面的例子，来理解一下 array 和 create_union 的用法，其他的读者可以自行练习：

```
hive> DESC studentScore;
OK
student              string
course               string
score                int
Time taken: 0.139 seconds, Fetched: 3 row(s)
hive> SELECT * FROM studentScore;
OK
xiaohong  CHINESE  85
xiaohong  MATHS  78
xiaohong  ART  99
lilei  CHINESE  89
lilei  ART  78
Time taken: 0.115 seconds, Fetched: 5 row(s)
hive> SELECT student, IF(array_contains(array("CHINESE","MATHS"), course),
"MAIN_COURSE", "OTHER"), create_union(if(score>80, 0, 1),"excellent",
"normal") FROM studentScore;
OK
```

```
xiaohong  MAIN_COURSE  {0:"excellent"}
xiaohong  MAIN_COURSE  {1:"normal"}
xiaohong  OTHER  {0:"excellent"}
lilei  MAIN_COURSE  {0:"excellent"}
lilei  OTHER  {1:"normal"}
Time taken: 0.174 seconds, Fetched: 5 row(s)
```

7. 函数

(1) 数学函数。

HiveSQL 中支持很多数学函数，如 log、ln、power、sqrt、sin、cos、e、floor、ceil 等。

(2) 集合操作函数。

size()可以返回 map 或者 Array 的大小。map_keys()、map_values()可以返回一个 map 中的所有键或者值的集合数组。Array_contains()可以用来判断一个数组中是否包含给定的元素。Sort_array()可以将给定的数组按照升序排列。

(3) 类型转换函数。

binary(string | binary)可以把参数转为 binary。可以使用 CAST(expr AS <type>)的方式做类型的转换。

(4) 日期函数。

最常用的日期函数是取出一个日期的年 year、取出月 month、取出日期 day、取出小时 hour、取出分钟 minute、取出秒 second。HiveSQL 中还支持比较日期 date_diff，日期增减 date_add、date_sub，返回当前日期 current_date、current_timestamp，对日期进行格式化 date_format 等操作。

(5) 条件操作函数。

条件操作函数主要有根据条件选择使用值 IF(testCondition, TrueValue, FalseValue)、CASE WHEN ELSE END，判断字段是否 NULL——isnull()、isnotnull()，返回第一个非空元素 COALESCE。

(6) 字符串操作函数。

字符串操作函数主要有：字符串连接函数 concat、concat_ws，编码解码函数 encode、decode，字符串长度 length，字符串翻转 reverse，截取字符串 substr，删除字符串边上的空白 trim 等。

9.5 Hive 的自定义函数

前面我们介绍了 Hive 中的 DDL 和 DML，其中，最常用的是查询语句 SELECT，而且 Hive 已经默认支持多种关系运算、算数运算、逻辑运算等操作，而且支持很多内置函数。

在实际业务场景中还是会遇到一些复杂的业务逻辑，会遇到 HiveSQL 的内置函数无法处理的情况。这时，就需要自己写一些自定义的函数(User Defined Function，UDF)来处理，自定义的 UDF 有三种。

(1) 普通 UDF：接收表中的一行数据，处理之后返回一行。

(2) 聚合 UDF(UDAF)：接收表中的多行数据，处理之后返回一行。

(3) 展开 UDF(UDTF)：接收表中的一行数据，处理之后返回多行。

由于 Hadoop、Hive 这一套框架是 Java 语言开发的，所以 UDF 都需要使用 Java 来做。下面详细讲解一下三种 UDF 的定义。

9.5.1 UDF

在做业务分析的时候，经常需要根据日志中的 IP 来获取用户的地域信息，进而做一些基于地域维度的业务判断。这个时候，就需要将 IP 转换为地域了。本小节给读者讲解一个将 IP 转为地域 UDF 的整个过程。经调研，选择纯真 IP 库(参考网址：http://www.cz88.net/)。当然，你也可以使用类似的其他 IP 地址库，但是应注意，每一家 IP 库的格式可能不太一样，需要根据地址库的格式来写出正确的解析程序，才能正常使用。本示例的代码在 czIpInfo 项目中，读者可自行部署和使用。

1. 核心业务逻辑——解析 IP 地址

我们要做的这个 UDF，是接收一个 IP 作为输入，通过在 IP 地址库文件中查找 IP 所在的地址段，来决定这个 IP 所在的地域和运营商，并将结果返回。

我们把网上下载到的 IP 地址库文件 qqwry.dat 放入项目的资源文件目录 resources 中。

接下来，我们要按照约定的格式，读取 IP 地址库，读者可以查看项目中的 IPSeeker.java 文件。IPSeeker 的 IP_FILE 定义了从 qqwry.dat 中读取 IP 信息的数据，之后，在构造函数里面使用了 IPSeeker.class.getClassLoader().getResourceAsStream 的方式来获取项目资源文件的 InputStream，之后，在 toRandomAccessFile 中，按照每兆读取一次的方法，获取 qqwry.dat 的内容，形成 this.ipFile 这个 RandomAccessFile 的句柄。如果获取不到文件或者读取出错，则直接抛出 RuntimeException 退出。

```
if (this.ipFile == null) {
    try {
        InputStream ipFileStream = IPSeeker.class.getClassLoader()
          .getResourceAsStream("qqwry.dat");
        this.ipFile = toRandomAccessFile(ipFileStream);
    } catch (Exception e) {
        throw new RuntimeException(
          "IP 地址信息" + IP_FILE + "文件没有找到，IP 显示功能将无法使用");
    }
}
```

在 getIPLocation 方法中，使用已经转为字节数组的 IP，在 locateIP 中，从前到后查找此 IP 在 qqwry.dat 中对应的记录，如果找到，则获取 offset 对应的 IP 的地域信息并返回：

```
public IPLocation getIPLocation(byte[] ip) {
    IPSeeker.IPLocation info = null;
    long offset = locateIP(ip);
    if (offset != -1L) {
        info = getIPLocation(offset);
    }
    if (info == null) {
        info = new IPSeeker.IPLocation(this);
        info.country = "未知国家";
        info.area = "未知地区";
    }
    return info;
}
```

在 getLocation 方法中，我们可以获取一个 IP 所属的地域，对于已经查找过的 IP，我们会把这个 IP 和对应的地域放入到 ipCache 中，以便于再次查找的时候可以快速返回。

其他读取 IP 地址库的部分不详细讲解，读者可以自行查阅资料。

2. 传入 IP 地址的适配

有些时候，我们传入的 IP 地址是一个用英文句号分隔的 4 个 1~255 的数字组成的字符串，如“202.110.34.234”。有的时候，我们传入的是一个长整型数字，如 3396215530(这个数值与前面的字符串表示的是相同的 IP，读者可以自行研究)。但是，我们上面的 IPSeeker 只支持 public String getLocation(String ip)。因此，我们在 IPUtils.java 中给出了一个方法：public static String long2Ip(long ip)，这样，在 UDF 中就可以实现无论是传入字符串还是数值的 IP，都可以找出这个 IP 对应的地域了。

3. 完成 UDF 开发

UDF 需要继承 org.apache.hadoop.hive.ql.exec.UDF，并书写类的 evaluate 方法。

可以看到，在 IPInfoUDF.java 中，我们实现了 4 个 evaluate 方法。前两个是针对字符串的 IP，可以是直接返回通过纯真 IP 库解析出来的，也可以通过与 Contants.java 中的国家、省份、城市信息做匹配，来获取一个 IP 所述的国家、省份、城市。后两个 evaluate 方法是调用了 IPUtils.java 中的 long2Ip 方法，之后再使用前两个方法来获取结果。如果我们直接运行 IPInfoUDF.java 的 main 方法，则可以获取到如下结果：

```
202.110.34.234: 辽宁省盘锦市

辽宁省盘锦市
中国
辽宁
盘锦
```

4. 打包和部署

按照 maven 项目部署好之后，我们可以编译打出 JAR 包。然后，需要将这个 JAR 包加入到 Hive 里面。有两种方式：临时的 Hive 函数和永久的 Hive 函数。

(1)　如果只是临时需要使用的 Hive 函数，可以使用如下的方法，这种方法在断开当前的 Hive 连接之后，就无法再使用该函数了：

```
hive>add jar /home/work/ipinfo.jar;
Added [/home/work/ipinfo.jar] to class path
Added resources: [/home/work/ipinfo.jar]
hive>create temporary function getArea as 'com.hive.udf.IPInfoUDF';
OK
Time taken: 14.96 seconds
```

接下来，就可以使用这个函数了：

```
hive> select getArea(3396215530) from show where p_date='20150101' limit 1;
OK
辽宁省盘锦市
Time taken: 0.206 seconds, Fetched: 1 row(s)
hive> select getArea(3396215530,'country'),
getArea(3396215530,'province'),getArea(3396215530,'city') from show where
p_date='20150101' limit 1;
```

```
OK
中国  辽宁  盘锦
Time taken: 0.178 seconds, Fetched: 1 row(s)
```

(2) 第二种方式是创建一个长期的 Hive 函数：

```
[work@localhost:~]$ hadoop fs -put /home/work/ipinfo.jar /tmp/ipinfo.jar
[work@localhost:~]$ hive
Logging initialized using configuration in
jar:file:/home/work/lib/hive-common-1.2.1.jar!/hive-log4j.properties
hive> create function default.getArea as 'com.hive.udf.IPInfoUDF' using JAR
"hdfs://YOURCLUSTERNAME/tmp/ipinfo.jar";
converting to local hdfs://YOURCLUSTERNAME/tmp/ipinfo.jar
Added [/home/work/hivelogs/tmp_resources/work/17b89988-bbc0-40e9-a39d
-3901516a6fc4_resources/ipinfo.jar] to class path
Added resources: [hdfs://YOURCLUSTERNAME/tmp/ipinfo.jar]
OK
Time taken: 15.137 seconds
hive>select default.getArea(3396215530,'province') from show where
p_date='20150101' limit 1;
OK
辽宁
Time taken: 1.274 seconds, Fetched: 1 row(s)
```

应注意，在创建长期的 Hive 函数之前，先要确定函数的命名空间，我们上面直接把这个函数放在 default 下面了，所以后面使用的时候，也需要使用 default.getArea。另外，上面的 YOURCLUSTERNAME 在使用的时候，应替换为读者具体的集群名称。

与使用 Hive 表类似，我们也可以使用 show functions 来查看现有哪些函数，使用 drop function if exists functioname 来删除现有的函数。

5. 注意 UDF 优化

类似于上面的例子，是一个 IP 转地域的函数。如果要在一个很大的分区上执行这个操作(比如一天的数据量很大的情况)，对于每一行记录，都会执行这个 UDF，如果不做优化，每次执行都重新读取 qqwry.dat 数据。设业务数据对于相同的 IP 会出现多次，如果不做优化，则同一个 IP 出现几次就需要查找几次。这时候，我们需要针对这种情况做一些针对性的优化，以避免因为 UDF 的原因，拖慢整个表的查询速度。

比如，上面代码中会有 if (this.ipFile == null)的判断，避免多次读取，上面代码中也会有 ipCache，避免对于相同的 IP 做多次查找。

当然，对于 UDF 的书写，需要根据当时的业务情况来做针对性的优化，这里没有通用的规则，需要在理解 MapReduce 执行原理的情况下做。

小提示

> 对于 UDF 的效率，可以对比使用和不使用 UDF 的时候的查询效率，来判断，如果使用 UDF 后时间明显变长，则建议做优化。

9.5.2 UDAF

UDAF 即聚合函数，一般是在分组(或者整张表)的数据处理过程中使用的。比如，在一

个表中有几列：学生 id、班级、学科、成绩，在处理的时候，我们希望按照班级来分组，计算学生的平均成绩，或者按照学科来分组，计算平均成绩。这时候，都需要按照某一列做 GROUP BY，之后对成绩列做 AVG 操作来获取平均成绩。其中，求平均的操作 AVG 就是 Hive 中默认支持的一个聚合操作。

下面的网址中已经列出了默认支持的聚合函数的列表，类似 sum、max、count distinct 以及一些基本的数学运算，已经在 Hive 默认支持的列表里面：

```
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF#
LanguageManualUDF-Built-inAggregateFunctions(UDAF)
```

但是，由于不同应用场景要做的数据处理逻辑上千差万别，总还会有一些操作不在 Hive 的默认支持列表里面。比如，我们要对一个分组中的某些列数据做一些复杂的运算。

(1) 获取一列数据用逗号连接在一起的结果字符串，类似于 MySQL 中的 GROUP_CONCAT。

(2) 将一个数值列的所有值按照大小排序后，去掉两个最大值，去掉两个最小值，之后做平均。

(3) 取出分组中 A 列的数据符合条件 X 的行中，B 列数据的总和；如果没有符合条件的行，则赋予默认值。

在这些种类的分组运算中，由于逻辑不具有特别强的通用性，Hive 默认支持的聚合函数是不能完成功能的。但我们仍旧想使用 HiveSQL，就只能自己写 UDAF 去定义这些运算规则了。

Hive 支持两种类别的 UDAF：简单 UDAF 和通用 UDAF。简单 UDAF 书写简单，但是，由于采用了 Java 反射机制，所以性能上会有劣势，并且不支持类似变量长度这样的参数列表。通用 UDAF 对这些特性支持得好，但写起来会复杂一些。

下面，我们分别使用简单 UDAF 和通用 UDAF 来完成上面所说的第一个复杂运算操作，即分组内的字符串连接。

1. 简单 UDAF

本书提供了一个简单的 UDAF 示例，即 udafExample 项目中的 com.hive.udaf.Concat 类。

在实现简单 UDAF 的时候，代码逻辑需要满足如下几个条件。

(1) UDAF 类必须继承 org.apache.hadoop.hive.ql.exec.UDAF。

(2) UDAF 类需要有一个实现 org.apache.hadoop.hive.ql.exec.UDAFEvaluator 接口的静态内部类。

(3) 静态内部类需要实现 UDAFEvaluator 接口的 init、iterate、terminatePartial、merge、terminate 这几个函数。其中，init 函数用于做一些初始化的操作；iterate 接收传入的参数，并进行处理，返回是否正常处理完毕的结果；terminatePartial 是 iterate 函数执行的结果，返回当前处理过程的数据；merge 接收 terminatePartial 返回的结果，并做数据的 merge 操作；terminate 返回聚合函数最终的结果。

代码里面其他详细的情况，读者可以自行研究。书写完代码之后，也是与前面讲述的 UDF 添加到 Hive 的方法相同。添加完之后，就可以在 Hive 里面使用了：

```
hive> create temporary function getConcat as 'com.hive.udaf.Concat';
OK
```

```
Time taken: 0.029 seconds
hive> select getConcat(financestatus,"-") from financedata where
p_date=20160101 group by company_id limit 5;
OK
NORMAL-NORMAL-NORMAL-NORMAL-NORMAL-NORMAL
NORMAL-NORMAL-NORMAL-NORMAL-NORMAL-NORMAL-NORMAL
NORMAL-NORMAL-NORMAL-NORMAL-NORMAL-NORMAL-NORMAL-NORMAL-NORMAL
-NORMAL-NORMAL
NORMAL-NORMAL-NORMAL-NORMAL-NORMAL-NORMAL
NORMAL-NORMAL
Time taken: 32.515 seconds, Fetched: 10 row(s)
```

2. 通用 UDAF

开发通用 UDAF 有两个步骤：第一个是编写 resolver 类，它负责类型检查，如果你想做运算符重载，也可以在这里做；第二个是编写 evaluator 类，真正实现 UDAF 的逻辑。一般来说，顶层的 UDAF 类继承自 org.apache.hadoop.hive.ql.udf.GenericUDAFResolver2，如果读者希望能屏蔽 Hive 接口未来的变化，应该继承 AbstractGenericUDAFResolver。Evaluator 类一般是静态内部类。

一般来说，通用的 UDAF 的大体骨架如下面的代码示例。Log 对象用来写入警告和错误到 Hive 的 log；getEvaluator 根据 SQL 传入参数的类型，来返回合适的 Evaluator；然后就是根据具体需要支持的参数类别，预先定义对应的 Evaluator。

```
public class GenericUDAFHistogramNumeric extends
  AbstractGenericUDAFResolver {
    static final Log LOG =
      LogFactory.getLog(GenericUDAFHistogramNumeric.class.getName());

    @Override
    public GenericUDAFEvaluator getEvaluator(GenericUDAFParameterInfo info)
      throws SemanticException {
        //Type-checking goes here!

        return new GenericUDAFHistogramNumericEvaluator();
    }

    public static class GenericUDAFHistogramNumericEvaluator
      extends GenericUDAFEvaluator {
        //UDAF logic goes here!
    }
}
```

通用的 UDAF 的例子可以参考 udafExample 项目中的 com.hive.udaf.GenerateConcat 类，这个类中，与上文的简单 UDAF 类似，提供了连接一个字段的功能。

在 getEvaluator 方法中，我们做了限定，只允许传入 string 类型的参数。

在静态内部类 GenericConcatString 中，我们定义了一个 ConcatAgg 类，用来存放连接字符串的结果。Init、getNewAggregationBuffer、reset、iterate、terminatePartial、merge、terminate 方法都需要重载 GenericUDAFEvaluator 中的基础的定义。需要注意，terminate 方法返回的值必须是实现了 org.apache.hadoop.io.WritableComparable 接口的对象，否则，在运行 MapReduce 的时候，Hadoop 无法识别。关于其他的内容，读者可以自行研究。

打包、上传、添加函数的方法与简单 UDAF 相同。运行之后结果如下：

```
hive> create temporary function getGenericConcat as
'com.hive.udaf.GenericConcat';
OK
Time taken: 0.029 seconds
hive> select getGenericConcat(financestatus) from financedata where
p_date=20160101 group by company_id limit 5;
OK
NORMAL,NORMAL,NORMAL,NORMAL,NORMAL,NORMAL
NORMAL,NORMAL,NORMAL,NORMAL,NORMAL,NORMAL,NORMAL
NORMAL,NORMAL,NORMAL,NORMAL,NORMAL,NORMAL,NORMAL,NORMAL,NORMAL,
NORMAL,NORMAL
NORMAL,NORMAL,NORMAL,NORMAL,NORMAL,NORMAL
NORMAL,NORMAL
Time taken: 32.515 seconds, Fetched: 10 row(s)
```

更多的关于如何写好 UDAF 的介绍，可以参考如下地址：

```
https://cwiki.apache.org/confluence/display/Hive/GenericUDAFCaseStudy
```

9.5.3　UDTF

上面讲解的 UDAF 是将一张表或者经过 GROUP BY 之后的一个分组的数据加以处理的函数。本节要讲的 UDTF 则用来解决输入一行输出多行的问题。

要编写 UDTF，需要继承 org.apache.hadoop.hive.ql.udf.generic.GenericUDTF 抽象类，实现 initialize、process、close 三个方法。下面结合一个对字符串进行分隔的例子来做讲解。

我们要做的是这样的一个功能：将一个 key-value 形式存储的数据的内容解析出来，并分割成不同的行。key 和 value 中间是使用“:”分隔的，不同的 key-value 之间是使用“;”分隔的。比如 maths:98;chinese:87;music:73 表示的是一个学生每门课的成绩。当我们要计算学生的平均分的时候，或者每门课程的平均分的时候，需要将这些数据解析成“课程”-“分数”对，才能计算。下面我们来看 UDTF 是怎样实现的，假如我们完成一个 ExplodeString 的类来完成这项工作。

1. initialize

initialize 用来做变量的初始化，以及用户在使用函数之前的参数和结果的判断。这个方法的输入参数是 ObjectInspector，表示的是我们在 HiveSQL 中使用 explodeString 的时候传入的参数；方法的返回值是 StructObjectInspector，表示用户调用 explodeString 的返回值。它与抽象类 GenericUDTF 一样，initialize 可以抛出 UDFArgumentException。

我们在 initialize 中判断参数值符合两个条件，一是参数个数只能是一个(这个字段里面存储的就是要分隔的字符串)；二是参数必须是基本的 String 类型。

其中，UDFArgumentLengthException 是 UDFArgumentException 的子类。

函数返回的只有两个 String 类型的字段，我们把它们叫作 col1 和 col2。

最后通过 getStandardStructObjectInspector 方法来获取 StructObjectInstpector 对象，作为函数的返回值。

```
@Override
public StructObjectInspector initialize(ObjectInspector[] args)
  throws UDFArgumentException {
    if (args.length != 1) {
        throw new UDFArgumentLengthException(
```

```
            "ExplodeString takes only one argument");
    }
    if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE) {
        throw new UDFArgumentException(
            "ExplodeString takes string as a parameter");
    }
    ArrayList<String> fieldNames = new ArrayList<String>();
    ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();
    fieldNames.add("col1");
    fieldOIs.add(
      PrimitiveObjectInspectorFactory.javaStringObjectInspector);
    fieldNames.add("col2");
    fieldOIs.add(
      PrimitiveObjectInspectorFactory.javaStringObjectInspector);
    return ObjectInspectorFactory.getStandardStructObjectInspector(
             fieldNames, fieldOIs);
}
```

2. process

process 方法就是实际完成 UDTF 工作内容的方法。它的输入参数 args 就是 UDTF 在实际运行时的数据的值。但是，它的没有返回值，那它的结果是怎么输出的呢，可以看到下面代码中有一行 forward，就是在遇到需要输出的一行结束的时候，就是用 forward 传递给框架作为 UDTF 的输出。与父类一样，可以抛出 HiveException。

在 process 的业务逻辑里面，我们做的事情就是按照 “;” 来将字符串作为第一级分隔，然后对于分隔的所有结果，按照 “:” 作为第二级分隔。并将每个第二级分隔的 result 当作一行的结果，forward 返回给框架。

```
@Override
public void process(Object[] args) throws HiveException {
    String input = args[0].toString();
    String[] test = input.split(";");
    for (int i=0; i<test.length; i++) {
        try {
            String[] result = test[i].split(":");
            forward(result);
        } catch (Exception e) {
            continue;
        }
    }
}
```

3. close

在 close 方法中，对需要清理的内容进行清理。我们这个示例中的 close 方法体是空的。

更详细的内容，读者可参考 udafExample 项目中的 com.hive.udtf.ExplodeString 类。

下面我们对这个 UDTF 加以测试。添加 JAR 包和函数的过程与上面一样，运行结果如下所示。我们看到，当尝试给 explodeString 多个输入参数的时候，它会给出提示。

```
hive> create temporary function explodeString as
'com.hive.udtf.ExplodeString';
OK
Time taken: 0.029 seconds
hive> load data local inpath '/home/work/opdir/abc' into table allScores
partition(p_date='20160101');
```

```
Loading data to table test.allscores partition (p_date=20160101)
Partition testdb.allscores{p_date=20160101} stats: [numFiles=1, numRows=0,
totalSize=100, rawDataSize=0]
OK
Time taken: 1.391 seconds
hive> select * from allScores;
OK
xiaoming  maths:98;chinese:87;music:73  20160101
xiaohong  maths:85;chinese:69;sports:99  20160101
lilei  arts:95;maths:59  20160101
Time taken: 0.365 seconds, Fetched: 3 row(s)
hive> select explodeString(student,scorelist) as (col1,col2) from allScores
where p_date='20160101';
FAILED: UDFArgumentLengthException ExplodeString takes only one argument
hive> select explodeString(scorelist) as (col1,col2) from allScores where
p_date='20160101';
OK
maths  98
chinese  87
music  73
maths  85
chinese  69
sports  99
arts  95
maths  59
Time taken: 0.177 seconds, Fetched: 8 row(s)
```

另外，需要注意，在使用 UDTF 函数的时候，不能添加其他字段一起使用。如果需要多个字段的话，可以使用 lateral view，如下：

```
hive> select student, explodedScores.course, explodedScores.score from
allScores lateral view explodeString(scorelist) explodedScores as course,
score where p_date='20160101';
OK
xiaoming  maths  98
xiaoming  chinese  87
xiaoming  music  73
xiaohong  maths  85
xiaohong  chinese  69
xiaohong  sports  99
lilei  arts  95
lilei  maths  59
Time taken: 0.265 seconds, Fetched: 8 row(s)
```

因此，计算一个学生的平均分，就可以这样做：

```
hive> select student, avg(score) from (select student, explodedScores.course,
explodedScores.score from allScores lateral view explodeString(scorelist)
explodedScores as course, score where p_date='20160101') a group by student;
Query ID = work_20160422233302_ff830a00-dd6a-493c-977e-75bf174fa586
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1461381026752_0001, Tracking URL =
http://localhost:8088/proxy/application_1461381026752_0001/
```

```
Kill Command = /home/work/hadoop-2.6.2/bin/hadoop job  -kill
job_1461381026752_0001
Hadoop job information for Stage-1: number of mappers: 1; number of reducers:
1
2016-04-22 23:33:23,320 Stage-1 map = 0%,  reduce = 0%
2016-04-22 23:33:34,864 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 1.32
sec
2016-04-22 23:33:48,367 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU
3.17 sec
MapReduce Total cumulative CPU time: 3 seconds 170 msec
Ended Job = job_1461381026752_0001
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 3.17 sec   HDFS Read: 10338
HDFS Write: 52 SUCCESS
Total MapReduce CPU Time Spent: 3 seconds 170 msec
OK
lilei  77.0
xiaohong  84.33333333333333
xiaoming  86.0
Time taken: 48.098 seconds, Fetched: 3 row(s)
```

9.6　Hive 的高级使用

9.6.1　视图

在传统数据库中(如 MySQL)，会允许用户建立视图，简单理解，就是以一定的视角来查看数据。视图的结构与原始表的结构不相同，但是，视图的数据不做真实的存储，只是为了方便用户通过某种视角来使用数据。

Hive 里面的视图(VIEW)与传统数据库中的视图类似，在 Hive 0.6 版本及以上都可以使用。Hive 的视图具有如下特点。

(1)　VIEW 是逻辑存在，Hive 暂不支持物化视图。一般来说，当有查询使用到 VIEW 的时候，Hive 会执行对应的 SQL 来生成结果集，以备进一步查询(实际上，Hive 会把 VIEW 的语句和针对它的查询放在一起，来进一步处理)。

(2)　VIEW 只读，不支持 LOAD、INSERT、ALTER 这些对数据的修改操作。在需要改变 VIEW 定义的时候，可以使用 ALTER VIEW。

(3)　Hive 支持迭代视图。

1. 创建视图

创建视图的语法如下：

```
CREATE VIEW [IF NOT EXISTS] [db_name.] view_name [(column_name [COMMENT
column_comment], ...)]
  [COMMENT view_comment]
  [TBLPROPERTIES (property_name = property_value, ...)]
  AS SELECT ...;
```

VIEW 自创建那一刻开始，其定义就被固定下来，而不能再改变了。

VIEW 内可以包含 ORDER BY、LIMIT 子句，假如一个针对 VIEW 的查询也包含这些语句，则 VIEW 中的语句优先级高。例如，定义 VIEW 数据为 LIMIT 10，针对 VIEW 的查

询为 LIMIT 20，则最多返回 10 条数据。

在 Hive 0.13.0 以上的版本中，在 VIEW 的 SELECT 子句中，可以使用一个或者多个公用的表达式。

下面是一个创建视图的例子，这里是为了方便地使用 business 类型的用户信息，通过 JOIN 的方式建立了一个视图：

```
CREATE VIEW IF NOT EXISTS business_user AS
SELECT people.* FROM user JOIN cart
ON (cart.user_id=user.id) WHERE cart.type='business';
```

2. 删除视图

删除视图的语法如下：

```
DROP VIEW [IF EXISTS] [db_name.] view_name;
```

如果被删除的视图已经被其他视图引用了，Hive 不会给出警告，直到用户发现依赖它的视图也只能被删除了为止。

3. 修改视图

Hive 允许修改视图的基本属性：

```
ALTER VIEW [db_name.]view_name SET TBLPROPERTIES table_properties;
```

其中，table_properties 是由多个 key-value 对组成的(property_name = property_value, property_name = property_value, ...)。例如：

```
ALTER VIEW shipments SET TBLPROPERTIES
('created_at' = '2016-01-01 00:00:00');
```

在 Hive 0.11 以上的版本中，运行修改视图关联的 SELECT 子句。修改了视图关联的 SELECT 子句，就相当于修改了视图的定义。这个语法与创建视图(CREATE VIEW 或者 CREATE OR REPLACE VIEW)的效果类似。语法如下：

```
ALTER VIEW [db_name.]view_name AS select_statement;
```

更多关于 Hive 视图的介绍参见如下地址：

```
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#
LanguageManualDDL-Create/Drop/AlterView
```

9.6.2　索引

在 Hive 0.7 以上的版本中，支持在一张表上建立索引；在 Hive 0.8.0 以上的版本中，可以使用位图索引。

在 Hive 表上建立索引的目标是改善基于一张表的某些列的查询性能，比如，对于条件 ‘WHERE tab1.col1 = 10’，如果没有索引，则执行 SQL 的时候，需要全表扫描，如果有了基于 col1 的索引，则只需要加载和处理需要处理的那部分数据即可。当然，因为创建索引后需要对表中原始的数据做预处理，所以，这里会有预先处理的开销以及存储空间的开销。

1. 创建索引

在 Hive 0.13 以上的版本中，索引的名称都是大小写敏感的，但是推荐使用全小写。

在一张 Hive 表上创建索引的语法如下：

```
CREATE INDEX index_name
  ON TABLE base_table_name (col_name, ...)
  AS index_type
  [WITH DEFERRED REBUILD]
  [IDXPROPERTIES (property_name=property_value, ...)]
  [IN TABLE index_table_name]
  [PARTITIONED BY (col_name, ...)]
  [
     [ROW FORMAT ...] STORED AS ...
      | STORED BY ...
  ]
  [LOCATION hdfs_path]
  [TBLPROPERTIES (...)]
  [COMMENT "index comment"];
```

如果指定了 WITH DEFERRED REBUILD 语句，则索引创建时是空的，之后可以通过“ALTER INDEX ... REBUILD”在一个 partition 上或所有 partition 上构建索引。INDEX 的 partition 默认和表的 partition 是一致的，如果指定了 partition by 子句，则相当于指定了这个表的 partition 列的子集，其中，每个 partition 列的所有值都将建立索引。

在视图上不能创建索引。

可以通过 STORED AS 或者 STORED BY 来指定索引的存储格式，当然，某些必须使用某种格式的索引除外。

2. 删除索引

删除索引的语法如下：

```
DROP INDEX [IF EXISTS] index_name ON table_name;
```

如果读者没有使用 IF EXISTS，但是希望在没有索引的时候不会报错，可以在配置文件中把 hive.exec.drop.ignorenonexistent 设置为 true。

3. 修改索引

修改索引的语法如下：

```
ALTER INDEX index_name ON table_name [PARTITION partition_spec] REBUILD;
```

这条语句会重新构建一个由 WITH DEFERRED REBUILD 声明的索引，或者重新构建一个先前已经创建好的索引，如果指定了 partition，则只有被指定的 partition 会被重新构建。

应该注意如下两点。

(1) 当 Hive 数据更新时，必须调用该语句更新索引。

(2) index rebuild 操作是原子操作，因此，rebuild 失败时，先前的索引也无法使用了。

9.6.3 权限

Hive 的授权机制并不是绝对安全的，它的目的只是保证正常使用的用户不会误操作，

但并不能防止恶意用户钻空子。如果要使用 Hive 的权限管理功能，需要在 hive-site.xml 中设置如下两个选项：

```
<property>
   <name>hive.security.authorization.enabled</name>
   <value>true</value>
   <description>
      enable or disable the hive client authorization
   </description>
</property>
<property>
   <name>hive.security.authorization.createtable.owner.grants</name>
   <value>ALL</value>
   <description>
      the privileges automatically granted to the owner whenever a table
      gets created.
      An example like "select,drop" will grant select and drop privilege
      to the owner of the table
   </description>
</property>
```

hive.security.authorization.createtable.owner.grants 默认是 NULL，建议将其设置成 ALL，这样，用户才能够访问自己创建的表。

Hive 授权的核心就是用户、组、角色。

如果相对三个用户分别赋予数据库 1、数据库 2、数据库 1 和 2 的权限，则做法可以如图 9-4 所示。设置两个角色：角色 X 和角色 Y，分别对应数据库 1 和数据库 2 的权限。然后设置组 A 和组 B，并把用户 A 和用户 B 分配到组 A 和组 B。之后，设置角色 Z，指向角色 X 和角色 Y，之后设置组 C，并把用户 C 设置为组 C 中的用户。

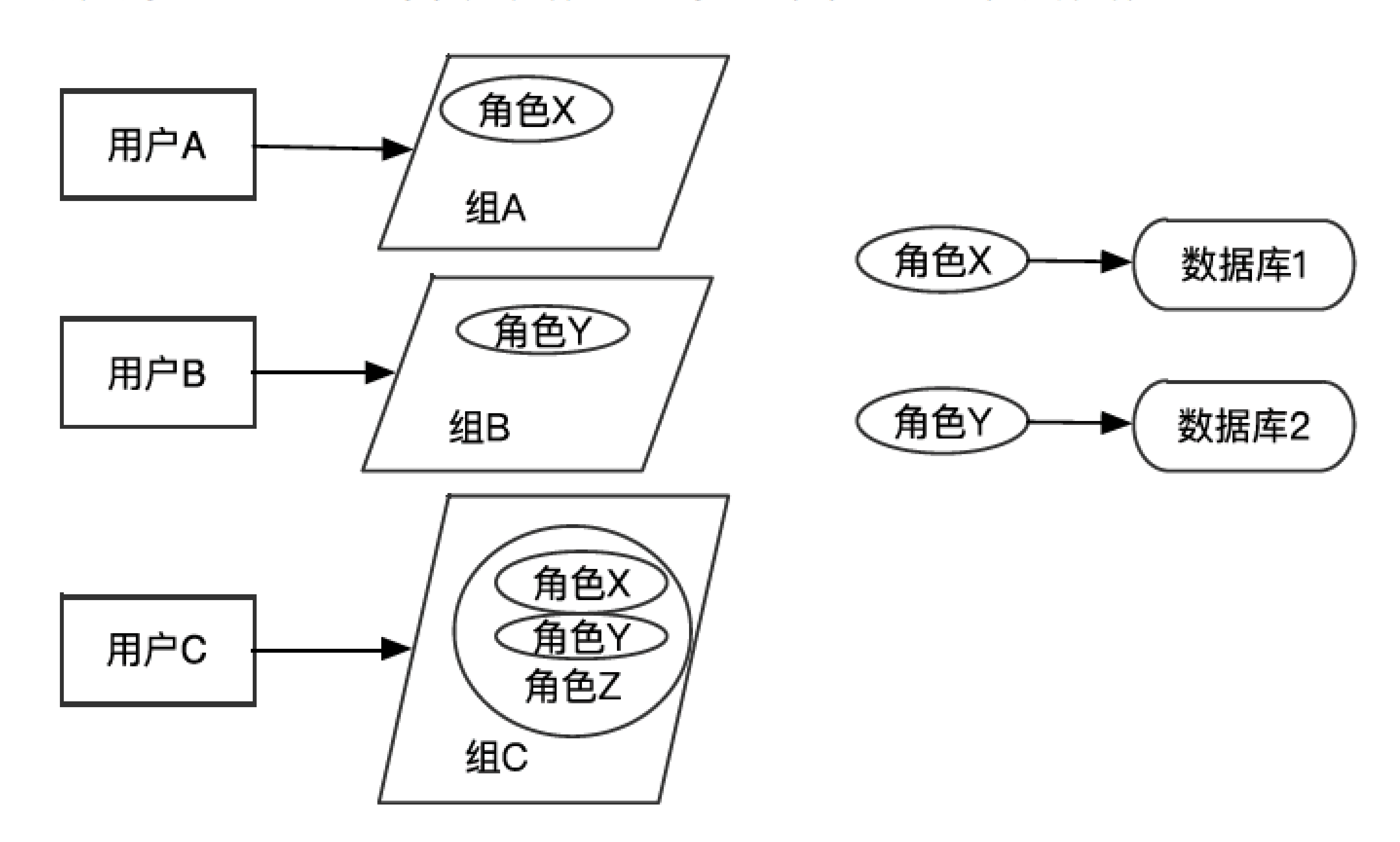


图 9-4 Hive 的授权

在 hive-site.xml 中，可以配置属性 hive.security.authenticator.manager，它就是管理用户和组的一个接口，默认使用 org.apache.hadoop.hive.ql.security.HadoopDefaultAuthenticator(如果我们要自定义鉴别方式的话，都需要实现这个接口)来做用户以及组权限的判定。当客户端连接 Hive 元数据库执行一个查询请求的时候，元数据库就会确定发送过来请求的用户和他所在的组，之后对比执行请求需要的权限和用户具有的权限，就可以决定这个用户是否有权限访问相应的元信息。元数据库对比权限的时候，先会判断执行 Hive 操作需要的权限，只要下面几项中满足一种即可：

- 用户被赋予了这个权限。
- 用户所属的任何一个组具有这个权限。
- 用户具有的角色，或者用户所属任何一个组具有的角色具有这个权限。

需要注意，Hive 只是控制了元数据的权限，是否具有 HDFS 的权限，是 Hadoop 的权限系统控制的。如下这种情况可能会发生：一个用户发出了操作请求，他具有 Hive 元数据的权限，但没有 HDFS 的权限。如果 HDFS 上的文件被手动修改过，但修改的时候没有关注权限，就有可能会出现这种情况。

创建、删除角色的方法如下：

```
CREATE ROLE role_name
DROP ROLE role_name
```

授予角色、取消角色、查看角色的方法如下：

```
GRANT ROLE role_name[, role_name] ... TO principal_specification[,
principal_specification] ...
[WITH ADMIN OPTION]
REVOKE [ADMIN OPTION FOR] ROLE role_name[, role_name] ... FROM
principal_specification[, principal_specification] ...
SHOW ROLE GRANT principal_specification
```

其中，principal_specification 可以是 USER user 或者 GROUP group 或者 ROLE role。使用 GRANT ROLE 方法给组分配权限的功能只有管理员才能操作。

在 Hive 里面，权限可以细分为如下这几类。

(1) ALL：所有权限。

(2) ALTER：允许修改元数据(modify metadata data of object)——表信息数据。

(3) UPDATE：允许修改物理数据(modify physical data of object)——实际数据。

(4) CREATE：允许进行 Create 操作。

(5) DROP：允许进行 DROP 操作。

(6) INDEX：允许建索引(目前还没有实现)。

(7) LOCK：当出现并发使用时，允许用户进行 LOCK 和 UNLOCK 操作。

(8) SELECT：允许用户进行 SELECT 操作。

(9) SHOW_DATABASE：允许用户查看可用的数据库。

其中，最常用的是 ALL、SELECT、CREATE 这几类权限。同样，这些具体的权限都可以分配给用户、角色或组。

经过以上的讲解，我们可以认为：在 Hive 的权限管理中，用户、角色和组都是权限的集合。更多详细的内容，读者可以查看：

```
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#
LanguageManualDDL-Create/Drop/Grant/RevokeRolesandPrivileges
```

9.6.4 Thrift 服务

Thrift 是 Facebook 开发的一个软件框架，它用来进行可扩展且跨语言的服务的开发，Hive 集成了该服务，能让不同的编程语言的远程客户端连接 Hive。目前的 Hive Thrift 服务是改进的 HiveServer 服务，支持多个客户端的并发以及认证。本来的理想设计是提供类似

于 ODBC 或者 JDBC 类的客户端 API 服务。

到现在为止，已经有很多用户报告了 Hive Server 在并发方面的 Bug。使用现在的 Thrift 机制来支持并发的连接是不现实的，所以 Hive 并不提供服务端的会话保持机制。因此，Hive 也就不能区分连接请求属于哪个已经连接的客户端的请求。现在有一些在这方面的努力，感兴趣的读者可以阅读下面的链接：

```
https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Thrift+API
```

1. Thrift 服务配置

在 hive-site.xml 中可以配置如下的变量。

- hive.server2.thrift.min.worker.threads：最小的工作线程数，默认是 5。
- hive.server2.thrift.max.worker.threads：最大的工作线程数，默认是 500。
- hive.server2.thrift.port：监听的 TCP 端口，默认是 10000。
- hive.server2.thrift.bind.host：绑定的 TCP 地址。

可以使用环境变量来覆盖 hive-site.xml 中的值：HIVE_SERVER2_THRIFT_BIND_HOST 和 HIVE_SERVER2_THRIFT_PORT，分别可以用来指定 TCP 地址和监听端口。

2. 启动 Thrift 服务

启动 Thrift 的服务很简单，有下面两种方式：

```
$HIVE_HOME/bin/hiveserver2
$HIVE_HOME/bin/hive --service hiveserver2。
```

可以使用--hiveconf <property=value>这种方式来指定配置参数值，例如：--hiveconf hive.server2.thrift.port=10001。

3. beeline 访问

beeline 是一个 JDBC 客户端，它有嵌入模式和远程模式两种工作模式。在嵌入模式中，它的工作方式与 Hive 命令行类似，在远程模式中，它可以连接一个单独的 Thrift 服务。可以在这里找到它：$HIVE_HOME/bin/beeline。

启动 beeline 之后，可以通过如下方式连接 Thrift 服务：

```
beeline> !connect jdbc:hive2://<host>:<port>/<db>;auth=noSasl hiveuser pass
org.apache.hive.jdbc.HiveDriver
```

比如：

```
beeline> !connect jdbc:hive2://localhost:10000
Connecting to jdbc:hive2://localhost:10000
Enter username for jdbc:hive2://localhost:10000: work
Enter password for jdbc:hive2://localhost:10000:
0: jdbc:hive2://localhost:10000 (closed)> show databases;
+---------------------+
| Database            |
+---------------------+
| default             |
+---------------------+
1 rows in set (0.00 sec)
```

beeline 连接了 Thrift 服务后，就可以正常地执行 HiveSQL 了。

对于 beeline 的使用，本书不做详细介绍，感兴趣的读者可以自行学习。

4. Java 代码访问

Thrift 服务很适合 Java 编程人员通过 jdbc 接口去访问 Hive，例如下面的代码：

```
TTransport transport = new TSocket("192.168.1.1", 10000);
TProtocol protocol = new TBinaryProtocol(transport);
ThriftHive.Client client = new ThriftHive.Client(protocol);

transport.Open();
client.execute("select * from pageview where dt = '2012-09-10' limit 10");
Console.WriteLine("the result is:");

var items = client.fetchAll();
foreach (var item in items)
{
   Console.WriteLine(item);
}
transport.Close();
Console.ReadLine();
```

当然，一般我们在 Java 项目中都会有一个通用的通过 JDBC 接口执行 Hive 操作的基础类。这也是我们自动调度数据平台的基础。

9.7　使用 Hive 构建数据仓库

在第 8 章中，我们一起学习了数据仓库的基础知识，读者对它有了基础的了解。而 Hive 就是构建数据仓库的常用框架，目前已经成为各大公司构建数据仓库的事实标准。本节以一个抽象但具有事实意义的小例子，来给读者讲解数据仓库的常见构造方法。

9.7.1　原始数据和结构

假如我们面对的是这样一个业务：一个移动应用在推广出去之后，用户数和数据量都有明显的增多，每天都产生了大量的日志在 HDFS 中。数据从业务产品流入到日志中的办法主要有：使用开源框架或者商业数据平台打日志、使用实时数据传输和接收框架来传递和接收日志(参见本书实时数据部分)等，这里，我们不做详细的讨论。

在这个移动应用中，会有多种客户端操作的类别，如安装应用、卸载应用、打开应用、查看应用的某个页面、点击应用中的某个位置、在某个地方停留了多长时间、应用崩溃信息等。具体要打印哪些发生在 APP 中的事件以及事件的属性，是根据业务发展的状态，以及产品和分析需求来设计的。我们在这个练习中，只关注两个事件：用户启动和点击页面，并且只关注这两个事件中非常有限的几个事件的属性，另外，会有一张从业务 MySQL 库里面获取的数据表。

(1) 在用户启动事件中，我们只关注：用户的设备 ID，即 User Device ID，简称 udid；注册用户的 ID，即 User ID，简称 uid；用户使用的 APP 版本，简称 app_version；用户手机的操作系统，如 Android、iOS 等。

(2) 在用户点击事件中，我们只关注：点击页面的 udid 和点击页面的 URL。

(3) 由于分析的时候，我们会很关心在不同级别的注册用户的表现(可能会根据用户付费，或者停留时间等，给用户划分级别)，因此，我们需要从业务 MySQL 库里面获取注册用户和它的级别的信息。

我们虚构的数据信息已经在附件的压缩包中，读者可以自行获取，之后，读者可以详细读一下数据的内容，然后我们可以将数据按照如下的方式加载到 Hive 中，假如说我们已经把数据放在了/tmp/dwdata 目录下，接下来把数据 LOAD 到 app_dw 库中。

```
USE app_dw;
CREATE TABLE IF NOT EXISTS dw_launch_raw(
  udid STRING,
  uid STRING,
  app_version STRING,
  os STRING)
PARTITIONED BY (p_date STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n';

CREATE TABLE IF NOT EXISTS dw_click_raw(
  uid STRING,
  url STRING)
PARTITIONED BY (p_date STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n';

CREATE TABLE IF NOT EXISTS reg_user_info(
  uid STRING,
  user_level int)
PARTITIONED BY (p_date STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n';

LOAD DATA INPATH '/tmp/dwdata/dw_launch_raw' INTO TABLE dw_launch_raw
PARTITION(p_date=20160101);
LOAD DATA INPATH '/tmp/dwdata/dw_click_raw' INTO TABLE dw_click_raw
PARTITION(p_date=20160101);
LOAD DATA INPATH '/tmp/dwdata/reg_user_info' INTO TABLE reg_user_info
PARTITION(p_date=20160101);
```

加载数据完成后，读者可自行验证，是否三张表里面的数据都已经有了。

注意，我们每张表都会使用 p_date 作为分区字段。因为 date 可能会在某些框架中作为关键字，故在每次使用的时候，都需要使用反引号括起来，导致使用繁琐，所以我们直接在前面添加 p_作为前缀，表明是一个 PARTITION 字段。另外，对于日志表，一般都会有至少一个分区字段来对不同日期的数据加以分隔，还有些业务场景会使用多个分区字段来分隔不同的数据内容。值得注意的是：对于 reg_user_info 这张表来说，并不是日志，实际上，它是一个维度表(或者叫配置表)，来自业务库。这里可能还会涉及一个叫“缓慢变化维”的问题，读者可自行阅读本书前面数据仓库部分的理论，对于其中的解决方式也可以根据业务场景有不同的解决方案，我们这里简单地使用 p_date 来做不同日期的分割。

注意，上面是为了方便我们做练习，直接将数据加载到 Hive 库中，实际的应用场景中，并不一定是通过这种方式加载的。有可能是通过一个单独开发的模块写入的，也有可能是通过 Sqoop 这种框架将 MySQL 的数据导入 Hive，具体的数据入库方式，根据数据仓库工具、所在组织的规范，以及业务场景的不同，会有不同的情况。这里我们无法详尽阐述。

9.7.2 数据需求和模型设计

在这个示例中，我们期望能获取每个级别的 launch 用户的总用户数和总点击情况。因此，我们设计的结果表有用户级别、用户数、点击情况三个非分区字段。

```
CREATE TABLE IF NOT EXISTS dwr_sum_info(
  user_level int,
  user_count bigint,
  click_count bigint)
PARTITIONED BY (p_date STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n';
```

为了汇总出这个结果，我们可以直接从原始的 raw 日志和 reg_user_info 表关联得到，但是细心的读者可能会发现，在我们给出的日志数据里面会有很奇怪的现象：

- dw_launch_raw 数据中，第一个字段出现很多 NULL 的情况。
- dw_launch_raw 数据的第二个字段有很多为 0。
- dw_launch_raw 数据中，大部分数据行的第 4 个字段是 android 或者 ios，会有个别行这个字段是 Android。
- dw_launch_raw 数据中会有多个重复的行。
- dw_click_raw 里面会有个别行的第二个字段(即点击的页面)是空的。

值得庆幸的是，从 MySQL 库中导入进来的 reg_user_info 数据暂时没有发现问题。

为什么会发现数据有这么多问题呢？因为每个业务的上线运行，都会依赖于很多工程师开发的后台模块，模块之间的关系会比较复杂，模块运行的网络情况、设备情况也是多种多样的，并且，当用户量巨大的时候，我们不可预知数据传输过程会偶发什么问题。因此，在数据仓库中，很大的一部分经历都是在做数据清洗，做数据规范化。

但是应注意，对于 dw_launch_raw 的第二个字段为 0，表示是注册用户的 ID 为 0，这是属于正常的，因为并不是所有的用户都是注册用户。但是，第一个字段为空的话，则表示没有办法定位一个设备，则是不正常的情况。

因此，对于这个应用场景，我们给出了如图 9-5 所示的数据模型。其中，最下层是我们加载好数据的那一层，最上层是用户需求的表。中间层就是我们加入的数据清洗层。

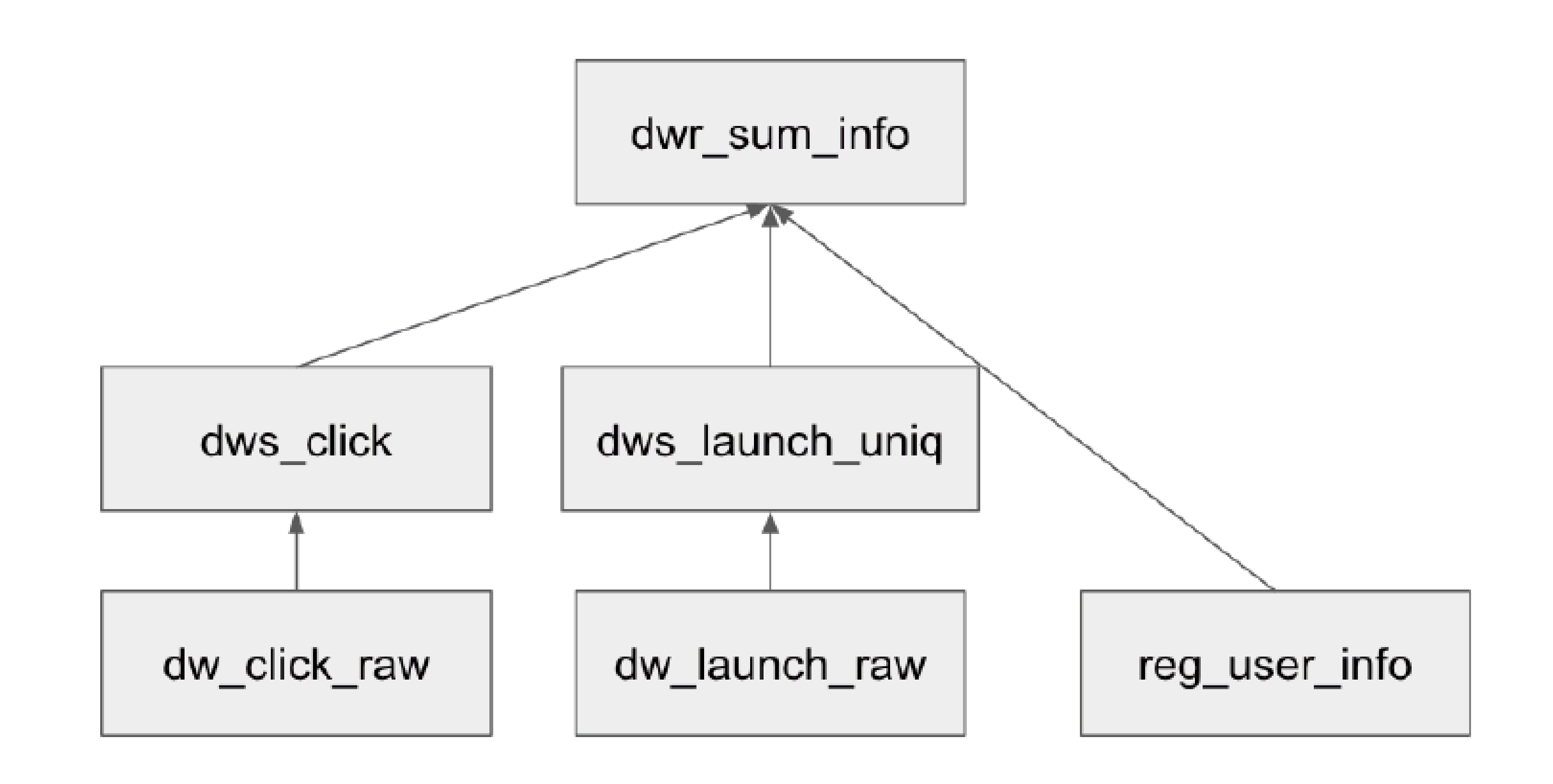


图 9-5 表之间的数据流向关系

有的读者可能会怀疑为什么这里中间的层次结构不是我们在数据仓库章节使用的“轻度汇总”？数据仓库章节里面，我们给出的是一套通用的层次结构设计，而遇到真实的业务场景的时候，需要根据需求的不同、数据的不同，对模型加以微调。这种微调需要更多地了解业务，也需要更多地了解数据，在此基础之上，才能设计出合理的模型。

这里我们给出的模型，主要是增加了两张中间表。

(1) dws_launch_uniq 是 dw_launch_raw 的一个清晰和简单的汇总，在这里，需要处理上面所说的几个问题，而且需要对 udid 做去重操作。

(2) dws_click 是 dw_click_raw 的一个简单处理，去掉不符合条件的记录。

这两张表的结构如下：

```
CREATE TABLE IF NOT EXISTS dws_launch_uniq(
  udid STRING,
  uid STRING,
  app_version STRING,
  os STRING)
PARTITIONED BY (p_date STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n';

CREATE TABLE IF NOT EXISTS dws_click(
  uid STRING,
  url STRING)
PARTITIONED BY (p_date STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n';
```

仔细看图 9-5，我们在对数据表做层次划分的时候，顺便对其命名做了规范，将数据分成了三个层次：dw 层、dws 层(表示汇总、summarize)、dwr 层(表示结果、result)；并且对去重的表在最后增加_uniq 做标识。在大型数据仓库的建设过程中，需要有明确的、统一的规范，对表和字段等的命名和使用方式加以规范，尽量见名知义，通俗易懂，便于给数据仓库的使用方提供更好的服务。

9.7.3　各层次数据的生成

有了上面的基础数据和模型设计，下面就是具体的实施了。我们直接把各表的实现方式列在下面了：

```
INSERT OVERWRITE TABLE dws_launch_uniq PARTITION (p_date="20160101")
SELECT
  udid,
  uid,
  app_version,
  lower(os)
FROM
  dw_launch_raw
WHERE
  p_date='20160101'
  AND udid is NOT NULL AND udid <> "NULL" AND udid <> ""
GROUP BY udid, uid, app_version, lower(os);
```

经过这样处理后，dws_launch_uniq 表一共有 8080 条记录。

```
INSERT OVERWRITE TABLE dws_click PARTITION (p_date="20160101")
```

```
SELECT
    udid,
    url
FROM
    dw_click_raw
WHERE
    p_date='20160101'
    AND url IS NOT NULL AND url <> "NULL" AND url <> "";
```

经过这样处理后，dws_click 表一共有 2699 条记录。

```
INSERT OVERWRITE TABLE dwr_sum_info PARTITION (p_date="20160101")
SELECT
    userinfo.user_level,
    count(distinct userinfo.uid),
    sum(CASE WHEN click.udid IS NOT NULL THEN 1 ELSE 0 END)
FROM
    (SELECT * FROM test.dws_launch_uniq WHERE p_date='20160101') launch
    JOIN (SELECT * FROM test.reg_user_info WHERE p_date='20160101') userinfo
ON (launch.uid = userinfo.uid)
    LEFT OUTER JOIN (SELECT * FROM test.dws_click WHERE p_date='20160101')
click ON (launch.udid = click.udid)
GROUP BY
    userinfo.user_level;
```

在上面的示例中，我们看到，launch 和 userinfo 是直接使用 JOIN 关联的，这是因为我们只需要保留能完全关联得上的。而与 click 是通过 LEFT OUTER JOIN 关联的，因为对于每一个用户，可能会有多次点击，都需要保留。但是，最终在 sum 点击数的时候需要对关联不上的刨除。

经过这样处理，我们得到如下结果。即，在 1、2、3 级别的注册用户分别有 955 个、488 个、524 个，他们分别带来了 394、197、252 次点击。应注意，到了这个步骤，我们的练习已经完成了，但实际工作中，需要对此步骤的数据质量做更深入的分析和监控。

```
hive> select * from dwr_sum_info WHERE p_date='20160101';
OK
1   955   394   20160101
2   488   197   20160101
3   524   252   20160101
Time taken: 0.632 seconds, Fetched: 3 row(s)
```

9.8 小　　结

Hive 是基于分布式数据仓库的框架。本章首先介绍了 Hive 的配置和体系结构，之后讲解了 Hive 命令行的使用，以及 Hive 中的数据类型。HiveSQL 是工作中用到最多的，也是数据仓库发挥价值的最重要的方式，本章重点介绍了 Hive 的数据定义语言和数据查询语言。Hive 的用户自定义函数(包括普通自定义函数、聚合函数、展开函数)是处理复杂 HiveSQL 问题的一种常见方法。Hive 的视图、索引、权限、Thrift 服务也是项目开发中经常使用的技术。本章最后给出了一个数据仓库的非常简单的例子，期望读者能从这个例子中理解数据仓库的基本建设思路和方法。通过本章的学习，希望读者能对 Hive 的各个方面能有基础的了解，并能重点掌握 HiveSQL、自定义函数以及使用代码访问 Thrift 服务的方法。

大数据实时计算篇

第 10 章

Storm 实时系统

学习目标

本章带领读者一起学习 Storm 实时处理系统，Storm 已在国内外多家机构、公司中使用，并获得了极高的评价。Storm 通常要搭配 Kafka 进行部署，本章也将重点介绍这两部分。本章主要以 Apache Storm、Apache Kafka 官方提供的资料为出发点，结合实际工程实践进行讲述。

通过对本章内容的学习，读者应能够掌握 Storm 实时系统的基本知识，能够结合 Kafka 编写具有产业领域通用性的 Topology 程序。

本章要点

- 大数据实时系统概述
- Kafka 分布式消息系统
- Storm 实时处理系统

10.1 大数据实时系统概述

随着移动互联网的快速发展，大数据时代到来了。批处理技术得到了全面发展，涌现出很多的分布式存储、计算框架，过去的 10 年是大数据处理变革的 10 年，如 HDFS、YARN、MapReduce、Spark、Hive 等以及一些相关的分布式技术，使得我们能处理海量数据。但遗憾的是，这些数据处理技术都不是实时的系统，它们设计的目的，是离线计算，而不能满足实时计算应用场景。

在产业领域，随着业务及技术的发展，企业产品、运营等部门对大数据流处理的需求变得更加强烈，如对新上线业务的实时监控，能最早发现问题并迅速做出响应；对实时要求比较高的实时推广，能根据用户的操作信息，迅速匹配模型，决策推广内容；对趋势进行实时的统计、分析，对常见统计指标进行实时统计，使业务任务能以秒粒度对数据趋势进行分析。

此外，产业巨头贡献的流处理技术，加速了实时领域的快速发展。

目前，比较流行的实时计算框架如图 10-1 所示。

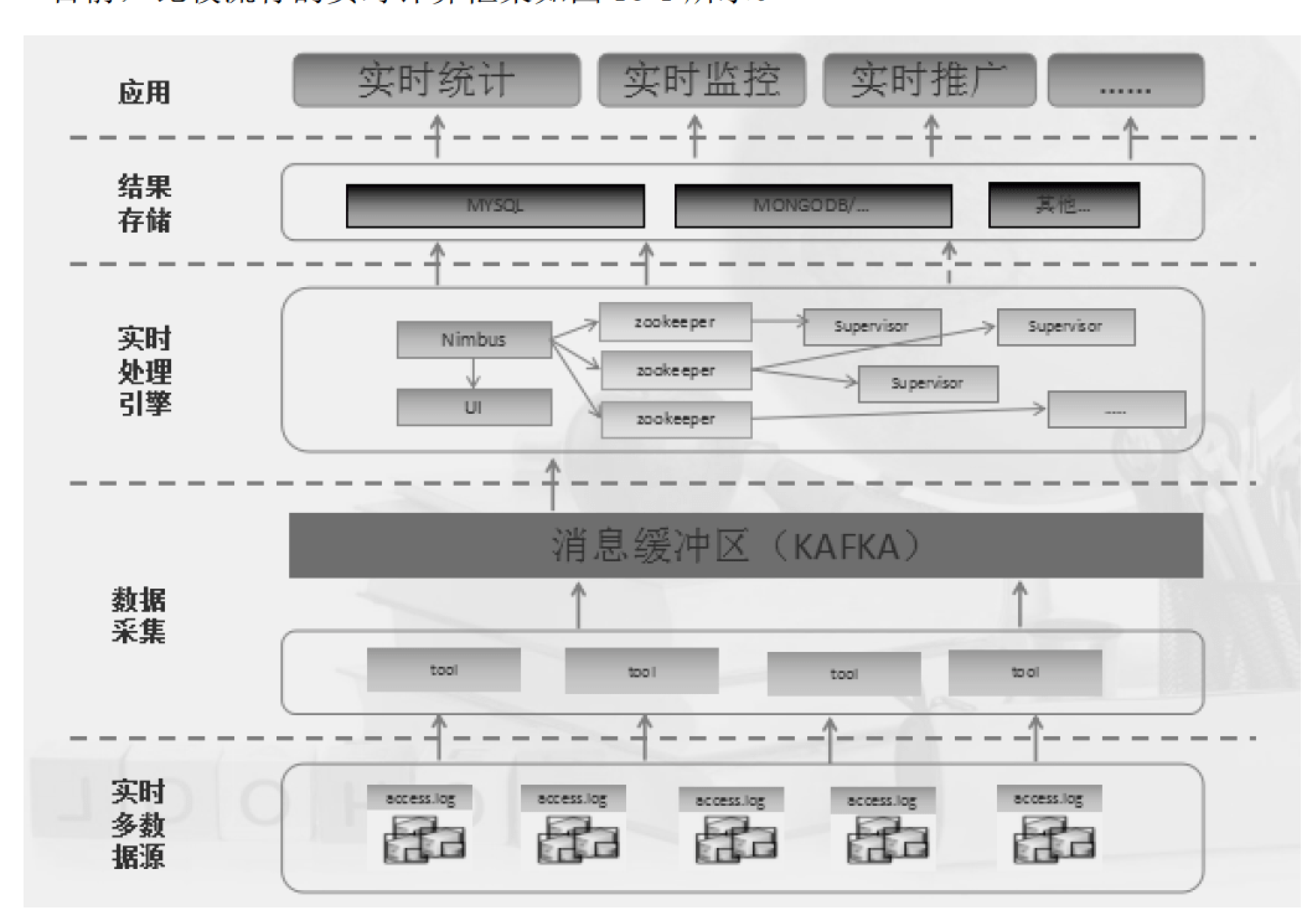


图 10-1 实时系统的架构

架构组成说明如下。

(1) 实时多数据源：实时产生的业务数据，常见的形式为日志、DB、互联网数据等。

(2) 数据采集：将业务数据经过 Flume 收集，并通过 ETL 处理后加载到消息队列中。

(3) 消息缓存：采用分布式消息系统 Kafka 作为缓存系统，接收来自采集工具的导入

数据，为实时分析引擎提供数据。

(4)　实时处理引擎：采用 Storm 开源工具，编写 Topology，实现业务的处理逻辑。

(5)　结果存储：对外提供应用的存储层，主要为关系数据库或 NoSQL 数据库或其他备份存储。

(6)　应用：基于流处理的应用，常见的有实时统计、实时监控、实时推广等。

本章后续内容将重点介绍消息缓存层 Kafka 分布式消息系统和实时计算引擎层 Storm 系统的原理、部署、运行。并分别附带实践代码，让读者熟悉产业领域如何以 Kafka、Storm 等软件构成一个能满足需求的大数据实时处理系统。

10.2　Kafka 分布式消息系统

Apache Kafka 是一种高吞吐量的分布式消息系统，可用于处理在线服务系统中的实时流数据。

10.2.1　Kafka 是什么

Kafka 是一个分布式的、分区的、带有副本机制的基于日志的高速消息缓存系统。它提供了一个消息系统的功能，但它的设计很独特，通过降低磁盘随机读写要求的磁头跳位频度，从而实现让磁盘的存储达到了内存速度的目的。Kafka 系统中的基本术语说明如下。

(1)　分类消息被称为主题。

(2)　发布消息到主题的进程是生产者。

(3)　订阅主题并且处理发布的消息的进程是消费者。

(4)　Kafka 集群包含一个或多个服务，每个服务叫 broke。

生产者(producer)通过网络发送消息到 Kafka 集群，消费者(consumer)从 Kafka 集群中消费消息，一个主题可以有多个生产者，也可以有多个消费者，如图 10-2 所示。

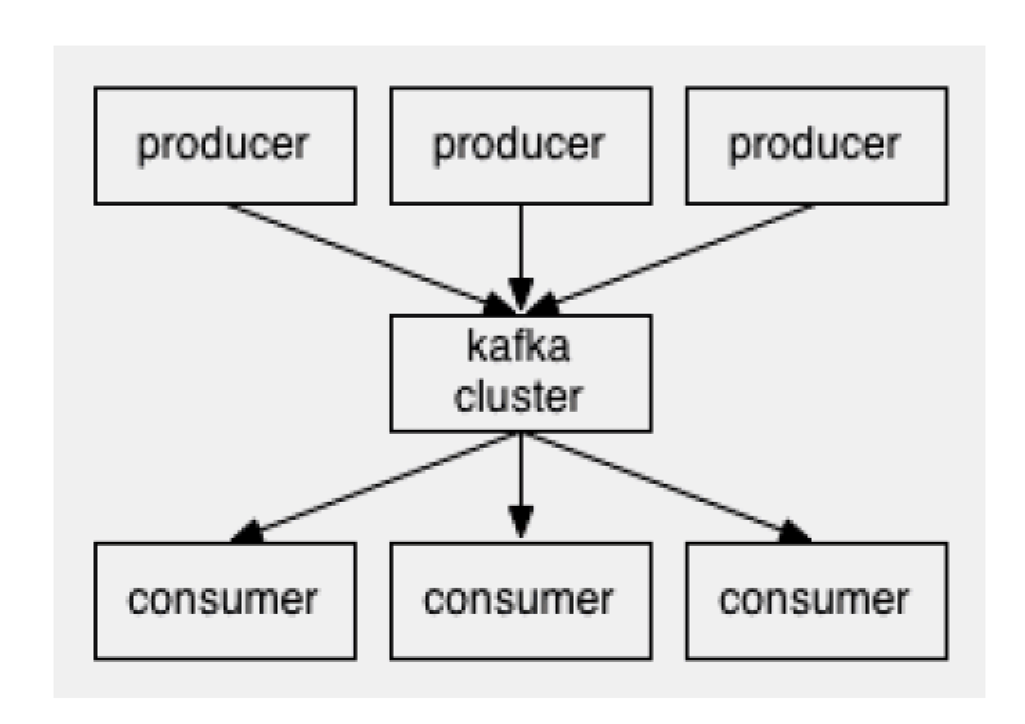


图 10-2　Kafka 生产者-消费者模型

在客户端、服务器之间的通信遵循简单、高性能、语言无关的 TCP 协议。

官网提供了 Kafka 的 Java 客户端，此外，Kafka 还支持多种语言的客户端。如果想不进行编程而直接使用 Kafka 的生产者-消费者服务，可以通过如下的方式：

```
# bin/kafka-console-producer.sh --broker-list cluster-node-01:9092 --sync
--topic topic_test
```

```
bin/kafka-console-consumer.sh --zookeeper cluster-node-01:2181 --topic
topic_test (从当前位置读)
bin/kafka-console-consumer.sh --zookeeper cluster-node-01:2181 --topic
topic_test --from-beginning  (从头最早数据读)
```

10.2.2 主题的工作原理

主题是消息发布的目的地，同时也是消费者的原地址。对于每个主题，Kafka 集群都保存了分区日志，以提升 Kafka 生产者、消费者的并行度，其形式如图 10-3 所示。

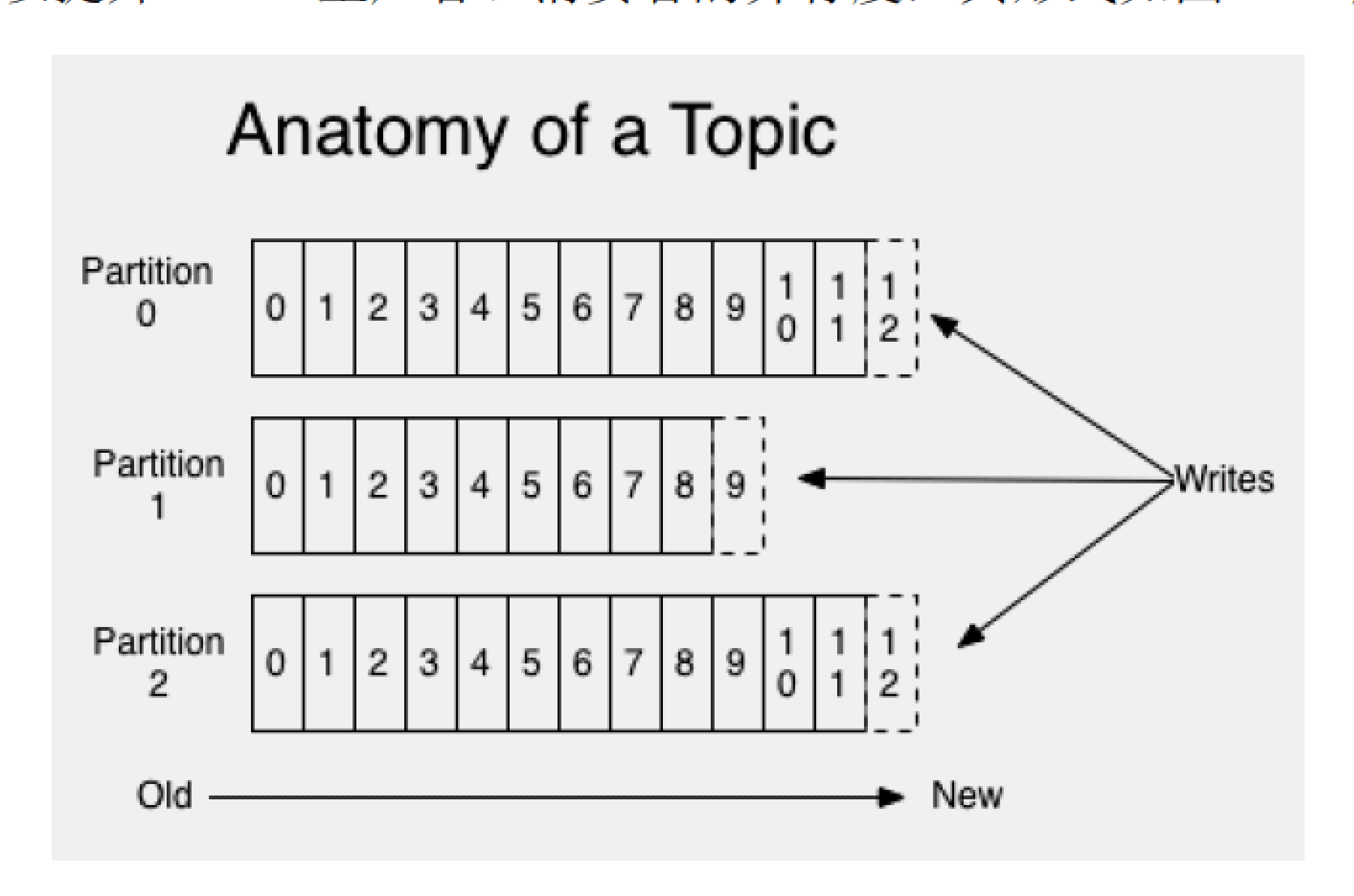


图 10-3　主题的分区

每个分区是独立有序的，消息队列递增追加，每个分区都以日志形式存储。每个分区中的每个消息都会被分配一个连续的 id 作为 offset，它可以唯一确定在分区中的消息。

Kafka 集群保存所有的发布消息，不管消息是否被消费。数据的生命周期是由配置文件中的属性决定的。例如，日志保留时间设置为 2 天，那么，一个消息从发布开始，两天内都是可用的，两天后，该消息被删除，配置文件 server.properties：

```
# The minimum age of a log file to be eligible for deletion
log.retention.hours=48
```

Kafka 系统可以高效地存储海量数据。每个消费者只需要保存元数据信息，即主题和偏移量。消费者控制它的偏移量，通常，消费者随着阅读线性增加偏移量，实际上，消费者可以从任何位置读取数据，例如，消费者可以重置已读过的偏移量，但不建议消费者随机读，而是相对稳定地顺序读。

这些功能的组合，意味着 Kafka 的消费者非常独立。一个消费者的来与走，不会对集群或者其他的消费者有任何影响。例如，你可以使用命令行工具查看主题的最新数据，同一主题的消费者的消费行为不会影响其他的消费者。

日志分区的两个目的：首先，允许横向扩展，每个独立的分区是一个独立的服务，一个主题可以有多个分区，所以能并行处理大量的数据。其次，分区作为最小的操作单元。

每个分区中的 offset 是有序的，生产者、消费者都是有序的。但是，消费者从多个分区中消费时，不能保证数据的原来顺序。如果要严格保证生产者的写入顺序也是消费输出顺序，只能在创建主题时设置分区数为 1，需要用户在速度和顺序之间做选择。

10.2.3 分布式分区

Kafka 系统中，日志的分区是分布式的，每个分区服务都处理数据和请求资源。每个分区有副本，副本数由配置属性确定。每个分区有一个服务，它作为操作的 leader，其他的分区副本作为 followers。由 leader 处理所有的对分区的读、写请求。而 follower 被动同步为 leader 的数据。如果 leader 失败，follower 中的一个将成为新的 leader。

10.2.4 生产者、消费者

生产者发布数据到指定的主题，生产者内部机制自动选择往主题的哪个分区分配数据。通过循环方式，简单实现加载均衡，或者可以通过一些语义分区函数来实现。

消息系统有两类模型：队列模型和“发布-订阅”模型。在队列模型中，多个消费者可以从服务读取消息；在“发布-订阅”模型中，消息是以广播形式传递给所有消费者的。

Kafka 提供了单一的消费者抽象概念——消费群，位于同一个消费群的消费者共用一个 offset，这也是一种提升并行处理能力的方案。

消费者用群名标记，每个消息发布到主题后，被传送到每个消费组的一个消费者实例中。消费者实例可以是独立的进程并在不同的机器上运行。

如果所有的消费者实例具有相同的消费群组，那么，这就像传统的队列模型，并能负载均衡。

如果所有的消费者实例拥有不同的消费群组，那么，这就像“发布-订阅”模型，并且每个消息被广播到所有的消费者。当然，这里，消费者必须要设置为消费该主题。

一个主题有少数量的消费群组，每个群组由许多消费者实例组成，消费者实例之间可扩展、高容错。

Kafka 系统中，订阅是一个群组，而不是一个单一的消费实例，结构如图 10-4 所示。

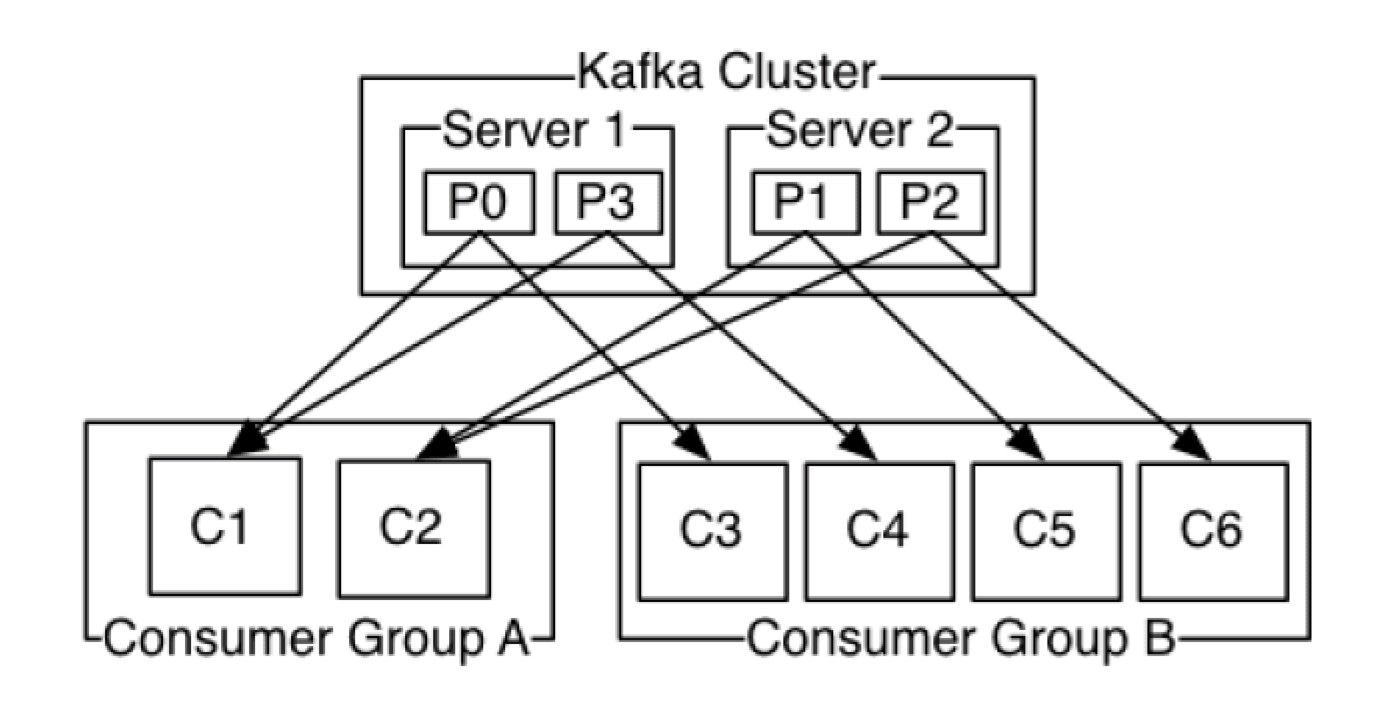


图 10-4 集群架构

一个两个节点的集群(两个 broker 进程)，某主题有 4 个分区，两个消费群组，其中，消费群组 A 有两个消费者实例，消费群组 B 有 4 个消费者实例。

Kafka 相对于传统的消费系统有更严格的顺序性。传统的消费队列有序地保存消息在服务上，当有多个消费者消费消息时，服务输出消息是有序的，严格按照存储时的顺序。然而，即使服务提供消息是有序的，消息被异步传送到消费者时，它们接收到的消息也可能

无序。为此，消息系统通常采用只有一个消费者来保证有序性，显然，这样就不能并行化处理。这在大数据领域中基本上是不可接受的。

在这方面，Kafka 做得更好。通过并行化的概念分区同一主题，Kafka 可以提供同一消费群中消费实例的顺序保证和负载均衡。通过分派主题的分区，对应消费群中的一个消费者，实现每个分发区只会被一个群中的一个消费者处理。这样，可以保证该消费者是唯一的，并且消费数据是有序的。注意：消费者的数量不要超过分区的数量，否则无意义。

前面已提到，Kafka 只提供一个分区的消息的有序性，不是主题中不同分区的消息的有序性。如果想得到主题全部消息的有序性，可以通过一个主题只有一个分区来实现，这也意味着每个消费群组中只有一个消费者实例。

10.2.5 数据保证

Kafka 的数据保证如下：

- 消息发送到主题的分区的顺序是发送的顺序。同一个生产者先发送 M1，又发送 M2，那么，在提交日志(commit log)中，M1 就会有较低的偏移量。
- 消费实例获取消息时，按照消息存储的顺序。
- 对于一个具有 N 副本的主题，可以容忍 N-1 个服务故障，而不会丢失消息数据。

10.2.6 Kafka 系统的应用场景

1. 消息

Kafka 是传统消息系统的有效替代品。消息队列的使用，通常是为了多种原因(降低生产者、消费者之间的耦合性，缓冲未处理的消息等)。相比于大多数消息系统，Kafka 有更好的吞吐量、内置分区、副本和容错机制，因此，Kafka 对于海量数据处理应用是一个极佳的解决方案。

2. 网站动态跟踪

Kafka 可以以实时的、“发布-订阅”模式重建网站的活动跟踪曲线。这表示可以将网站活动(PV、UV、搜索、其他用户行为)实时发布到主题中。通过订阅消息，应用到具体的业务中。常见的应用有：实时处理、实时监控、加载到 Hadoop 集群或者离线数据仓库系统、报表。动态跟踪对用户的浏览信息进行统计，在生产领域具有极高的应用价值。

3. 数值计算

Kafka 经常用于监控关键数据、关键指标，这需要从分布式应用中收集数据，经过计算引擎实现逻辑，并通过前端展示结果数据。

4. 日志聚合

很多人用 Kafka 作为日志聚合工具。日志聚合通常是指收集物理日志文件，并放到一个统一的中央位置以做全量处理。Kafka 抽象分拣的详细信息，给出了一个简洁的抽取日志后的时间数据作为流消息。这允许低延迟数据、对多数据源的支持、分布式的数据消费。相对于日志收集系统，如 Scribe 或者 Flume，Kafka 可以提供高性能对接接口，实现数据的

海量、高效接入。

5. 流处理

在很多应用场景中，用户需要做一些阶段性的数据处理，这些数据是从主题中消费的源数据。例如，一个处理工作流，需要爬取微博文章、放到 articles 的主题中；后面的处理可能将文章内容清洗、通用化并放入到另一个主题中，最后阶段可能是内容匹配。这展示了一个实时数据流的在多个独立 topic 之间的流向。Storm 和 Spark Streaming 是实现该类型处理的主流框架。

6. 事件采购

事件采购是把状态改变记录到按时间有序记录的序列。Kafka 支持非常大的日志存储，可以作为该类应用的优秀的后端服务。

7. 提交日志

Kafka 可以作为一个分布式的提交日志服务。日志在各个节点之间有备份。日志处理程序可以方便地从 Kafka 制定的主题中获取日志数据并进行处理。

10.2.7　Kafka 系统的部署

1. 账号相关

以 root 账号登录，创建 storm 账号：

```
[root@iZ25fnur5jkZ home]# useradd storm
[root@iZ25fnur5jkZ home]# passwd storm
```

切换为 storm 账号：

```
[storm@iZ25fnur5jkZ ~]$  su - storm
```

2. Zookeeper 安装

Kafka 的运行依赖 Zookeeper，用于记录 Kafka 相关的元数据信息。例如 topic 信息、消费者信息等。Zookeeper 可以是分布式集群，本书采用单点模式，单点向分布式扩展非常容易，只需要将{dataDir}目录下的 myid 配置为不同的数值，并添加新的{server.n}，启动，即为分布式 Zookeeper 集群。

(1) 下面安装单点模式的 Zookeeper。

① 创建 Zookeeper 目录、解压：

```
[storm@iZ25fnur5jkZ ~]$ mkdir zookeeper
```

② 进入 Zookeeper 目录：

```
[storm@iZ25fnur5jkZ ~]$ cd zookeeper
```

③ 从官方网站下载 Zookeeper 软件包，下载地址：

```
http://apache.fayea.com/zookeeper/zookeeper-3.4.6/
```

④ 解压安装包：

```
[storm@iZ25fnur5jkZ zookeeper]$ tar -xvf zookeeper-3.3.6.tar.gz
```

⑤ 修改 Zookeeper 的配置文件，需要修改配置文件中的 dataDir、dataLogDir、server.1。配置文件：

```
[storm@iZ25fnur5jkZ zookeeper]$ vim  conf/zoo.cfg
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
dataDir=/home/shaka/dep/zookeeper-3.3.6/data  (改成自己的目录)
# the port at which the clients will connect
clientPort=2181
# set logs
dataLogDir=/home/shaka/dep/zookeeper-3.3.6/logs  (改成自己的目录)
# set server
server.1=hostname:4887:5887   (改成自己的 hostname 或 IP)
#server.2=10.162.219.52:4887:5887
#server.3=10.163.15.119:4887:5887
# add by shaka
# set max client connects
maxClientCnxns=300
```

(2) Zookeeper 的基本操作。

① 启动 Zookeeper，赋权 bin 下所有文件的执行权限：chmod +x *。

```
[storm@iZ25fnur5jkZ zookeeper]$ bin/zkServer.sh start
JMX enabled by default
Using config:
/home/storm/streamprocessing/zookeeper/zookeeper-3.4.6/bin/../
conf/zoo.cfg
Starting zookeeper ... STARTED
```

② 启动后查看状态：

```
[storm@iZ25fnur5jkZ zookeeper]$ bin/zkServer.sh status
JMX enabled by default
Using config:
/home/storm/streamprocessing/zookeeper/zookeeper-3.4.6/bin/../
conf/zoo.cfg
Mode: standalone
```

③ 停止 ZK 服务：

```
[shaka@iZ25fnur5jkZ zookeeper]$ bin/zkServer.sh stop
JMX enabled by default
Using config:
/home/storm/streamprocessing/zookeeper/zookeeper-3.4.6/bin/../
conf/zoo.cfg
Stopping zookeeper ... STOPPED
```

④ 重启 ZK 服务：

```
[shaka@iZ25fnur5jkZ zookeeper]$ bin/zkServer.sh restart
JMX enabled by default
```

```
Using config:
/home/storm/streamprocessing/zookeeper/zookeeper-3.4.6/bin/../
conf/zoo.cfg
JMX enabled by default
Using config:
/home/storm/streamprocessing/zookeeper/zookeeper-3.4.6/bin/../
conf/zoo.cfg
Stopping zookeeper ... STOPPED
JMX enabled by default
Using config:
/home/storm/streamprocessing/zookeeper/zookeeper-3.4.6/bin/../
conf/zoo.cfg
Starting zookeeper ... STARTED
```

(3) ZK 的初始化目录为：/zookeeper/quota。Zookeeper 安装启动后，继续部署 Kafka。

① 从官网下载 Kafka，下载地址：

```
wgethttps://www.apache.org/dyn/closer.cgi?path=/kafka/0.8.1.1/
kafka_2.9.2-0.8.1.1.tgz
```

② 解压软件包：

```
[storm@hadoop-nn sbin]$tar -xvf kafka_2.9.2-0.8.1.1.tgz
```

3. 修改 Kafka 的配置文件

(1) 修改配置文件 vim conf/server.properties：

```
broker.id=0
host.name=yourhostname
zookeeper.connect=zookeeperhostname:2181 可逗号分隔配置多个
[storm@hadoop-nn sbin]chmod +x sbin/*
```

(2) 修改配置文件 vim log4j.properties：

```
log4j.appender.D.File =
/data1/home/shaka/kafka/kafka_2.9.2-0.8.1.1/logs/debug.log (改成自己的目录，
或者用相对路径)
log4j.appender.E.File =
/data1/home/shaka/kafka/kafka_2.9.2-0.8.1.1/logs/error.log (改成自己的目录，
或者用相对路径)
```

(3) 在 Kafka 目录下，创建 sbin 目录，存储常用的操作脚本。然后编写 Kafka 启动脚本 start-kafka.sh：

```
bin/kafka-server-start.sh config/server.properties
```

(4) 编写查看主题的脚本 list-topic.sh：

```
bin/kafka-topics.sh --list --zookeeper  zookeeperhostname:2181
(zookeeperhostname 改成 zookeeper 部署服务器的 hostname)
```

(5) 确保有执行权限，通过 svn、拷贝等方式可能会丢失执行权限，需要进行添加：

```
[storm@hadoop-nn kafka_2.9.2-0.8.1.1]$ chmod +x bin/*
[storm@hadoop-nn kafka_2.9.2-0.8.1.1]$ chmod +x sbin/*
```

(6) 启动 Kafka 服务：

```
sbin/start-kafka.sh
```

(7) 查看是否启动：jsp -l。

```
21689 kafka.Kafka (表示 Kafka 已经启动)
```

4. Kafka 操作实践

(1) 创建 topic 主题。主题的名称为 mykafka：

```
[storm@hadoop-nn kafka_2.9.2-0.8.1.1]$ bin/kafka-topics.sh --create
--zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic
mykafka
```

(2) 查看主题：

```
[storm@hadoop-nn kafka_2.9.2-0.8.1.1]$ sbin/list-topic.sh
topic-test-001
mykafka
Datacenter-001
```

可以看到，创建的主题已被列出，说明创建成功。测试用的主题 mykafka 采用一个分区，一个备份。

生产环境通常需要配置多个分区、多个备份，以提高 Kafka 的性能。

(3) 启动 consumer：

```
bin/kafka-console-consumer.sh --zookeeper hadoop-nn:2181 --topic mykafka
```

(4) 启动 producer：

```
bin/kafka-console-producer.sh --broker-list hadoop-nn:9092 --topic mykafka
```

(5) 在 producer 端输入，看 consumer 端的输出。

① 输入：

```
[storm@hadoop-nn kafka_2.9.2-0.8.1.1]$ bin/kafka-console-producer.sh -st
hadoop-nn:9092 --topic mykafka
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
hello world,kafka!
```

② 输出：

```
[storm@hadoop-nn kafka_2.9.2-0.8.1.1]$ bin/kafka-console-consumer.sh
--zookeeper hadoop-nn:2181 --topic mykafka
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
hello world,kafka!
```

5. Kafka 应用实践

模拟 Web 服务器实时产生日志，实时写入到 Kafka 中，并从 Kafka 中实时消费，写入到 MySQL 表中。

数据处理流程如图 10-5 所示。

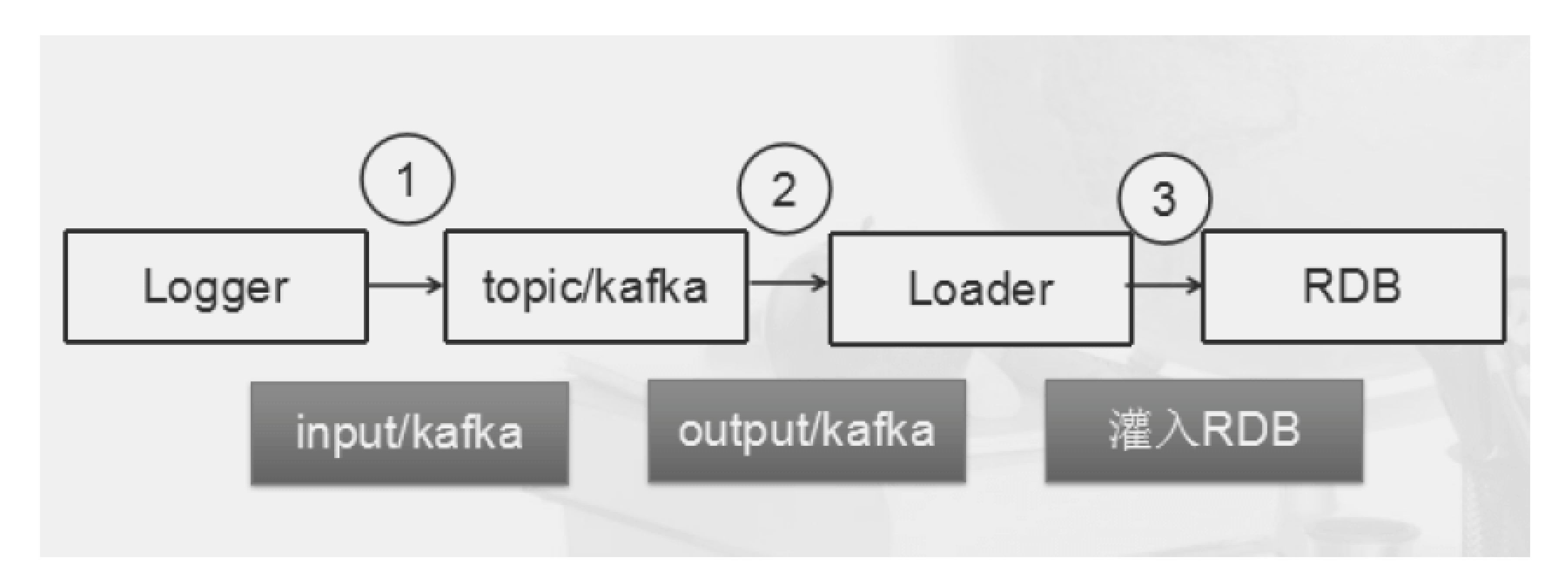


图 10-5　Kafka 数据收集流程图

Logger 为脚本代码，如下所示：

```
vim shelllogger.sh

#!/bin/sh
# start cmd:
# nohup sh shelllogger.sh  >> shelllogger.log 2>&1 &
# set timer
g_getTime=""
function getTime
{
    g_getTime=`date  '+%Y%m%d %H:%M:%S'`
}
#getTime && echo "[$g_getTime] [$0:$LINENO:$FUNCNAME] - "
# set function
function crawler
{
    int=1
    while(( $int<=1000000000 ))
        do
log="{ \"time_local\": \"01/Nov/2015:00:01:01 +0800\", \"remote_addr\":
\"182.92.77.57\", \"remote_user\": \"-\", \"body_bytes_sent\": \"5760\",
\"request_time\": \"0.005\", \"status\": \"200\", \"request\": \"GET
/jiayouserver/www/index.php\", \"request_method\": \"GET\",
\"http_referrer\": \"-\", \"body_bytes_sent\":\"5760\",
\"http_x_forwarded_for\": \"-\", \"http_user_agent\": \"Wget/1.12
(linux-gnu)\" }"
     let "int++"
     echo $log >> access.log
     sleep 1s
     #usleep 1000
     done
}
# main
crawler
```

Logger 打印 access.log 内容截取：

```
{ "time_local": "01/Nov/2015:00:01:01 +0800", "remote_addr": "182.92.77.57",
"remote_user": "-", "body_bytes_sent": "5760", "request_time": "0.005",
"status": "200", "request": "GET /jiayouserver/www/index.php",
"request_method": "GET", "http_referrer": "-", "body_bytes_sent":"5760",
"http_x_forwarded_for": "-", "http_user_agent": "Wget/1.12 (linux-gnu)" }
{ "time_local": "01/Nov/2015:00:01:01 +0800", "remote_addr": "182.92.77.57",
"remote_user": "-", "body_bytes_sent": "5760", "request_time": "0.005",
"status": "200", "request": "GET /jiayouserver/www/index.php",
```

```
"request_method": "GET", "http_referrer": "-", "body_bytes_sent":"5760",
"http_x_forwarded_for": "-", "http_user_agent": "Wget/1.12 (linux-gnu)" }
```

Input/kafka 将日志数据实时收集到 Kafka 中，主题为 mykafka：

```
[root@iZ25fnur5jkZ home] nohup tail -f ../../logger/access.log |
bin/kafka-console-producer.sh --broker-list hadoop-nn:9092 --topic
mykafka >logs/producer.log 2>&1 &
```

Output/kafka 从 mykafka 中读取数据：

```
[root@iZ25fnur5jkZ home]nohup
/usr/local/streamprocessing/kafka/kafka_2.9.2-0.8.1.1/bin/kafka-console-
consumer.sh --zookeeper hadoop-nn:2181 --topic mykafka | java
-Djava.ext.dirs=dist:lib loader.mysql.Loader > logs/consumer.log 2>&1 &
```

此处采用管道形式，将数据直接传递给写 RDB 的处理单元，将结果写入表中。

管道的工程对接代码如下：

```
/**
*  接收控制台的输入
*/
BufferedReader reader =
  new BufferedReader(new InputStreamReader(System.in));
String input = null;
while ((input=reader.readLine()) != null) {
    System.out.println(input);
    Loader.load(input);
}
reader.close();

//解析日志内容，一行日志，Json 格式，解析类
public class JsonLine {

    private Map<String, String> map = new HashMap<String, String>();
    public Map<String, String> et(String line) {
        try {
            map.clear();
            //日志原有字段
            map.put("time_local",   "-");
            map.put("remote_addr",    "-");
            map.put("remote_user",    "-");
            map.put("request",     "-");
            map.put("status",     "-");
            map.put("body_bytes_sent",    "-");
            map.put("http_referer",    "-");
            map.put("http_user_agent",    "-");
            map.put("http_x_forwarded_for", "-");
            //转化新字段
            map.put("location",    "-");
            map.put("date",      "-");
            //读出 Json 中的数据
            JSONObject logjson = JSONObject.fromObject(line);
            Iterator<String> it = logjson.keys();
            while (it.hasNext()) {
                String key = it.next();
                String value = logjson.getString(key);
                map.put(key, value);
            }
            //转化数据
            map.put("location", getLocation(map.get("remote_addr")));
```

```
            map.put("date", getDate(map.get("time_local")));
            System.out.println(map);
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
        return map;
    }
    public String getLocation(String remote_addr) {
        Ip2PlaceTool ipt = new Ip2PlaceTool();
        String location = ipt.getIpLocation(map.get("remote_addr"));
        return location;
    }

    public String getDate(String time) {
        // [19/Jun/2014:10:38:20 +0800]
        time = time.replace("[", "").replace("]", "").replace(" ", ":");
        StringTokenizer splitter = new StringTokenizer(time, "/:");
        // get day
        String day = null;
        if (splitter.hasMoreTokens()) {
            day = splitter.nextToken();
        }
        // get month
        String month = null;
        if (splitter.hasMoreTokens()) {
            JTimer timet = new JTimer();
            month = timet.MonthAto1(new String(splitter.nextToken()));
        }
        // get year
        String year = null;
        if (splitter.hasMoreTokens()) {
            year = splitter.nextToken();
        }
        // get date
        String date = year + month + day;
        return date;
    }

    public static void main(String[] args) {

        String line = "{ \"time_local\": \"16/Oct/2015:15:02:14 +0800\",
\"remote_addr\": \"211.157.142.222\", \"remote_user\": \"-\",
\"body_bytes_sent\": \"2102\", \"request_time\": \"0.003\", \"status\":
\"200\", \"request\": \"GET
/jiayouserver/www/index.php?factory=dream&user=%E5%B8%B8%E5%BF%97%E5%86%
9B HTTP/1.1\", \"request_method\": \"GET\", \"http_referrer\": \"-\",
\"body_bytes_sent\":\"2102\", \"http_x_forwarded_for\": \"-\",
\"http_user_agent\": \"Mozilla/5.0 (iPhone; CPU iPhone OS 8_0 like
Mac OS X) AppleWebKit/600.1.4 (KHTML, like Gecko) Mobile/12A365\" }";

        JsonLine ngix = new JsonLine();
        Map<String, String> res = ngix.et(line);

        System.out.println(res);
    }
}
```

写 MySQL 的主要代码如下：

```
//逐行进行解析
for(int i=0; i<rows.length; i++)
```

```
{
    //逐行进行解析
    JsonLine ngix = new JsonLine();
    Map<String, String> mapline = ngix.et(rows[i]);
    String date = mapline.get("date");
    String time = mapline.get("time_local");
    String ip   = mapline.get("remote_addr");
    String location = mapline.get("location");
    String status   = mapline.get("status");
    String body_bytes_sent = mapline.get("body_bytes_sent");
    String request = mapline.get("request");
    String http_referer = mapline.get("http_referer");
    String http_user_agent = mapline.get("http_user_agent");
    String http_x_forwarded_for = mapline.get("http_x_forwarded_for");

    // exec sql
    String sql = "insert into
t_nginx_access(date,time,ip,location,status,body_bytes_sent,request,http
_referer,http_user_agent,http_x_forwarded_for) values('" + date+ "',  '" +
time + "','" + ip + "','" + location + "','" + status + "','" + body_bytes_sent
+ "','" + request + "','" + http_referer + "','" + http_user_agent + "','"
+ http_x_forwarded_for + "');";
    System.out.println("sql=" + sql);
    ret = gmysql.exeSQL_DUI(sql);
}
```

10.3 Storm 实时处理系统

Apache Storm 是一个免费的、开源的、分布式的、实时的流计算系统，支持使用多种编程语言实现处理实时业务逻辑。

10.3.1 概述

大规模的实时数据处理已经越来越成为一种业务需求了，而缺少一个“实时版本的Hadoop”已经成为数据处理整个生态系统的一个巨大缺失。

Storm 应运而生，填补了这个缺失。Storm 出现之前，需要自己手动维护一个由消息队列(Queues)和消息处理者(Workers)所组成的实时处理网络，消息处理者从消息队列取出一个消息进行处理，更新数据库，发送消息给其他队列进行进一步处理，但是，这种计算方式的局限性太大：复杂、不健壮且扩展性差。

Storm 发布的第一天，Github 上的粉丝就超过了 1000 人。Storm 立刻登上了 Hacker News 网站的头条。2012 年 1 月做了一个调查，发现 Storm 已有 10 个产品用户，另有 15 个用户计划将要在他们的产品中使用 Storm，还有 30 家企业对 Storm 进行了试验。在发布后的 3 个月内，拥有了那么多的产品用户，这对于一个大型的基础型项目来说是非常有意义的。

在开源短短的三年后，Storm 于 2014 年 9 月 17 日正式步入顶级项目的行列。

10.3.2 为什么使用 Storm

Apache Storm 是一个免费的、开源的、分布式的、实时的流计算系统。Storm 使得流式数据处理变得非常容易，相对于 Hadoop 的批处理，Storm 是实时的流处理。Storm 简单、

可以使用多种编程语言实现处理逻辑。

Storm 有很多的使用案例：实时分析、在线机器学习、连续计算、分布式 RPC、ETL 等。Storm 速度快：每个节点、每秒钟以百万级的速度处理元组。Storm 易扩展、高容错、数据安全、易于安装和操作。

Storm 整合了消息队列和数据库技术。Storm Topology 以任意的复杂方式、分流各个阶段的计算结果来处理流数据，起点数据通常来源于消息队列，终点通常指向数据库技术。

Storm 系统在大数据实时领域被广泛使用，图 10-6 为部分使用单位。



图 10-6　Storm 系统的用户

Storm 集群主要由一个主节点(master node)和一群工作节点(worker nodes)组成，通过 Zookeeper 集群进行协调。

主节点通常运行一个后台程序——Nimbus，用于响应分布在集群中的节点，分配任务和监测故障。工作节点同样会运行一个后台程序——Supervisor，用于收听工作指派，并根据要求运行工作进程。每个工作节点都是 topology 中一个子集的实现。

10.3.3　Storm 系统的特点

1. 可扩展

计算任务可在多个线程、进程和服务器之间并行进行，支持灵活的水平扩展。

2. 高可靠

保证每条消息都能被完全处理。

3. 高容错性

Nimbus、Supervisor 都是无状态的，可以用 kill -9 来杀死 Nimbus 和 Supervisor 进程，然后再重启它们，任务照常进行。当 Worker 失败后，Supervisor 会尝试在本机重启它。

4. 支持多种编程语言

除了用 Java 实现 Spout 和 Bolt，还可用其他语言。

5. 支持本地模式

可在本地模拟一个 Storm 集群功能、进行本地测试。

6. 高效

用 ZeroMQ 作为底层消息队列，保证消息能快速被处理。

10.3.4 Storm 系统的工作机制

Storm 客户端提交 Topology 到 Nimbus，由 Nimbus 建立 Topology 的本地目录，通过 Zookeeper 分配资源和进行任务调度，监控 Task 心跳。

Zookeeper 存放公共数据，Nimbus 将分配给 Supervisor 的任务都写入到 Zookeeper 的 znode 中。

Supervisor 通过 Zookeeper 获得 Nimbus 分配的任务，并管理属于自己的 Worker 进程。Supervisor 获取 Task 后，启动任务 Worker，建立任务之间的连接。

流程如图 10-7 所示。

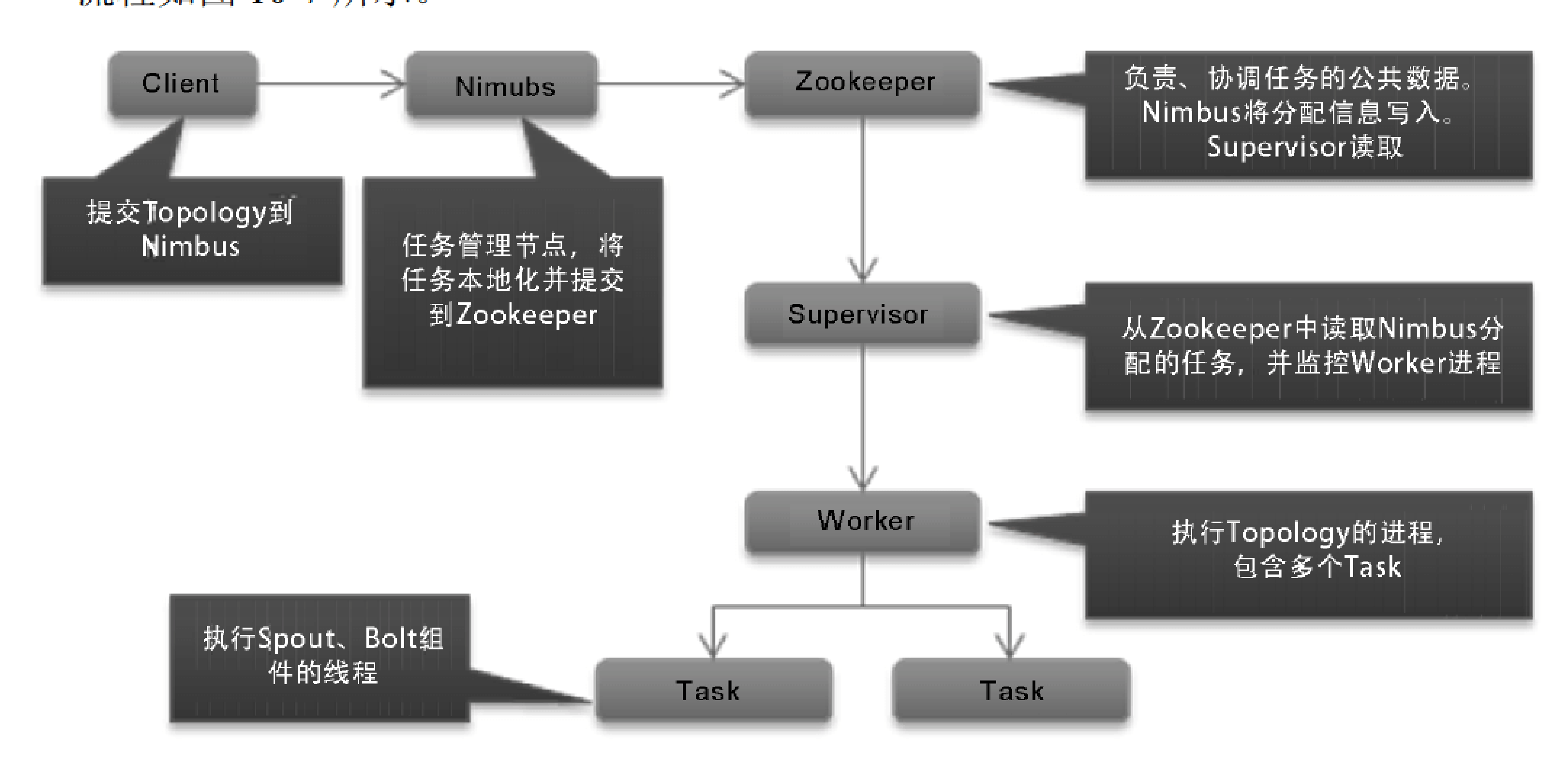


图 10-7 Storm 任务的流程

在 Storm 中，应用程序实现实时处理的逻辑被封装进 Storm 的 Topology 中。

一个 Topology 是由一组 Spout 组件(数据源)和 Bolt 组件(数据操作)通过 Stream Groupings 进行连接的图。

Spout：在一个 Topology 中产生源数据流的组件，从来源处读取数据并放入 Topology。

Bolt：在一个 Topology 中接收数据然后执行处理的组件。

Topology 的拓扑结构如图 10-8 所示。

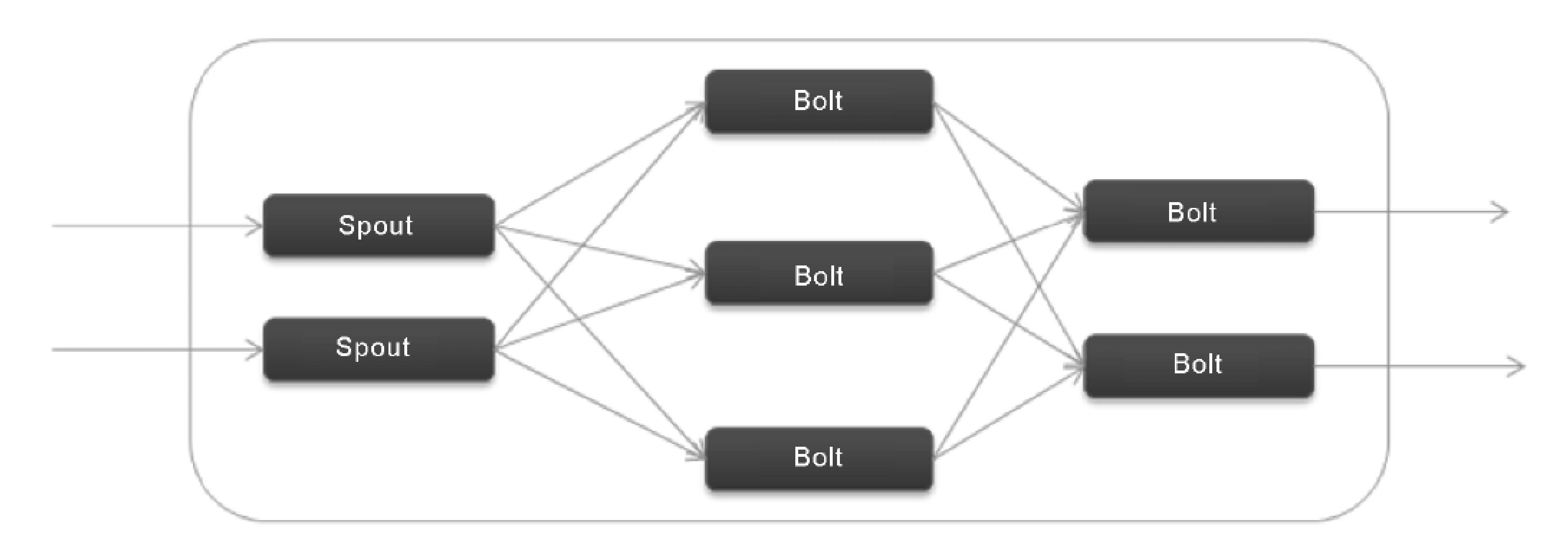


图 10-8　数据流向图(topology 的拓扑结构)

10.3.5　Storm 的分组方法

(1)　随机分组(Shuffle grouping)：随机分发 Tuple 到 Bolt 的任务，保证每个任务获得相等数量的 Tuple，如图 10-9 所示。

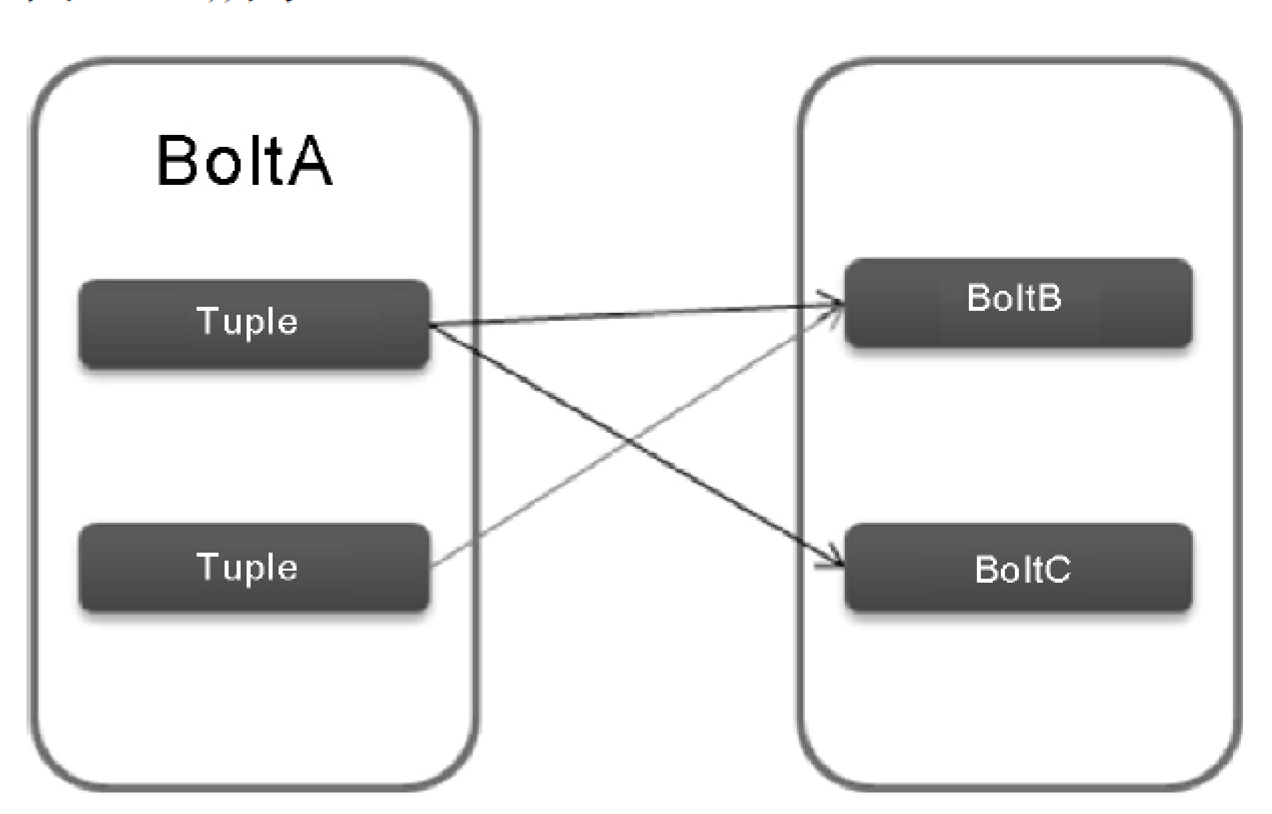


图 10-9　随机分组

(2)　字段分组(Fields grouping)：根据指定字段分割数据流，并分组。例如，根据 user-id 字段，相同 user-id 的元组总是分发到同一个任务，不同 user-id 的元组可能分发到不同的任务，如图 10-10 所示。

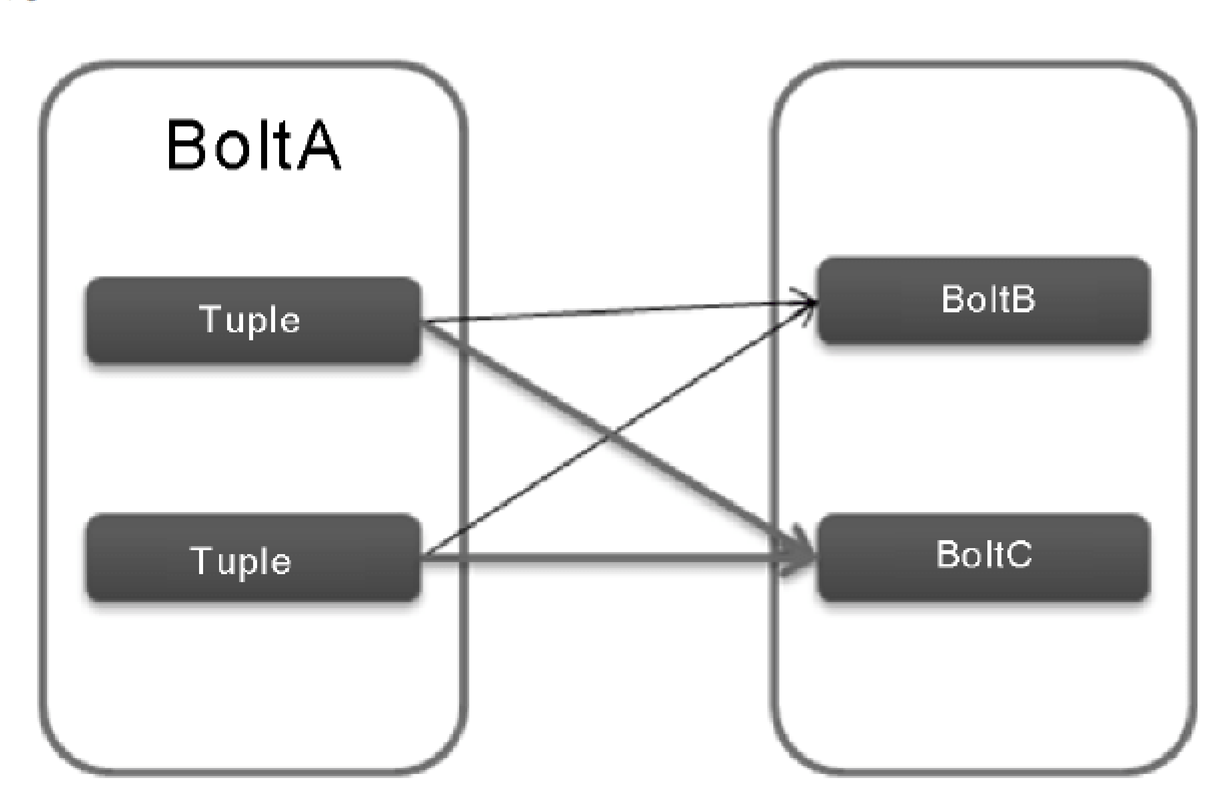


图 10-10　字段分组

(3) 全部分组(All grouping)：Tuple 被复制到 Bolt 的所有任务。这种类型需要谨慎使用，如图 10-11 所示。

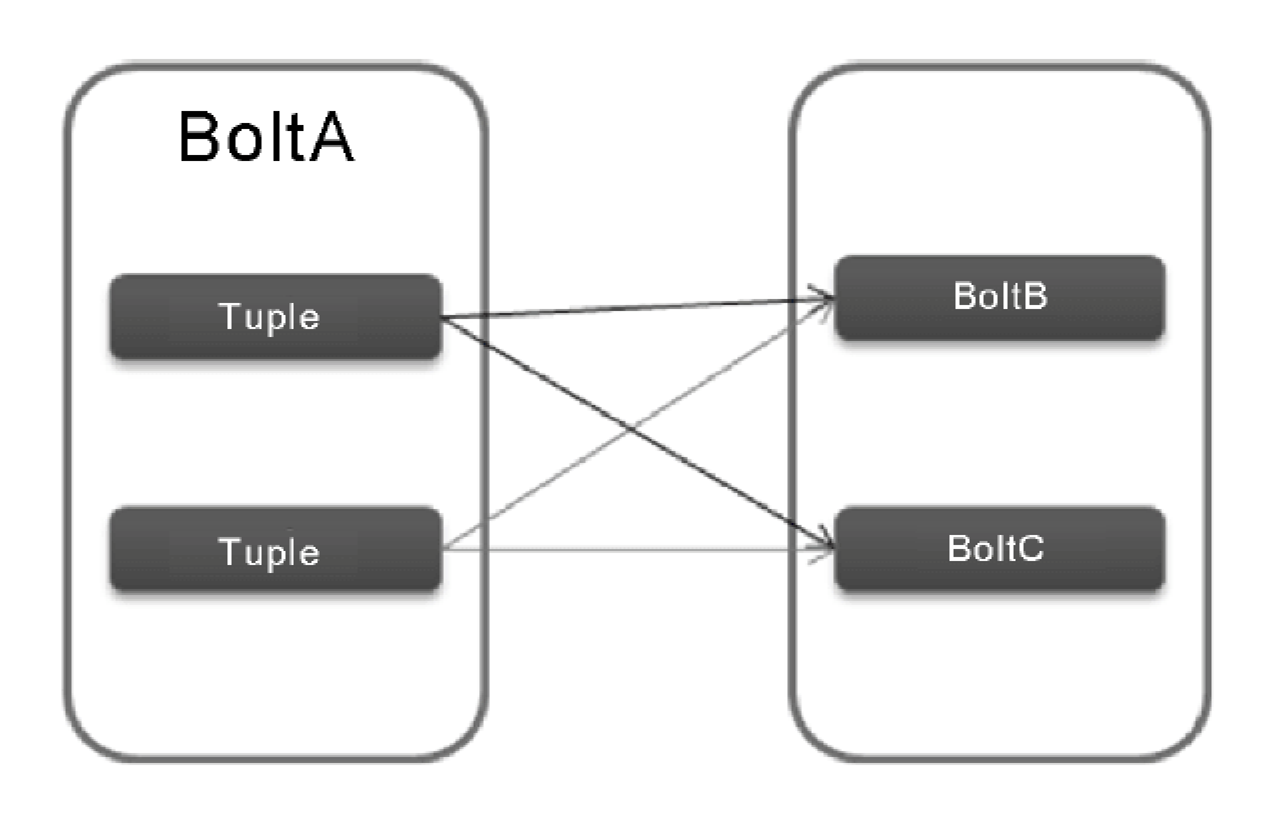


图 10-11 全部分组

(4) 全局分组(Global grouping)：全部流都分配到 Bolt 的同一个任务。

(5) 无分组(None grouping)：我们不需要关心流是如何分组的。目前，无分组等效于随机分组。但最终，Storm 将把无分组的 Bolts 放到 Bolts 或 Spouts 订阅它们的同一线程去执行(如果可能)。

(6) 直接分组(Direct grouping)：这是一个特别的分组类型。元组生产者决定 Tuple 由哪个元组处理者任务接收。

10.3.6 Storm 系统的组件

(1) Topology：Storm 中运行的一个实时应用程序。
(2) Nimbus：负责资源分配和任务调度。
(3) Supervisor：接受 Nimbus 分配的任务，启动和停止属于自己管理的 Worker 进程。
(4) Worker：运行具体处理组件逻辑的进程。
(5) Task：Worker 中每一个 Spout/Bolt 的线程称为一个 Task。
(6) Spout：在一个 Topology 中产生源数据流的组件。
(7) Bolt：在一个 Topology 中接收数据然后执行处理的组件。
(8) Tuple：一次消息传递的基本单元。
(9) Stream grouping：消息的分组方法。

10.3.7 搭建单点 Storm 系统

本部分介绍搭建 Storm 并运行 Storm 集群的步骤。

Storm 的版本：apache-storm-0.9.2-incubating。

下面是安装 Storm 集群的概括性步骤。

(1) 搭建 ZK 集群。
(2) 安装 Nimbus 和 Worker 机器上的依赖工具。
(3) 下载、解压 Storm 发行稳定包到 Nimbus 和 Worker 的机器上。
(4) 配置 Storm 的配置文件 storm.yaml。

(5) 启动 Storm 的 Nimbus 和 Supervisor 守护进程。

Storm 的依赖软件有 Python、Zeromq、Jzmq、Zookeeper。此外，系统应安装有 Java、Gcc、G++编译环境。

1. 安装 Storm 依赖包

在/home/storm/下：

```
[storm@iZ25fnur5jkZ dependence]$ mkdir dependence
[storm@iZ25fnur5jkZ dependence]$ more REAME
```

本文件夹的依赖文件仅为 Storm 依赖文件。

切换到 dependence 目录：

```
[storm@iZ25fnur5jkZ dependence]$  cd dependence
```

下载 zeromq、jzmq 到该目录。

安装 zeromq：

```
[storm@iZ25fnur5jkZ dependence]$ tar -xvf zeromq-4.0.4.tar.gz
[storm@iZ25fnur5jkZ dependence]$ cd zeromq-4.0.4
[storm@iZ25fnur5jkZ dependence]$./configure
[storm@iZ25fnur5jkZ dependence]$make
[storm@iZ25fnur5jkZ dependence]$sudo make install
```

安装 jzmq：

```
[storm@iZ25fnur5jkZ dependence]$cd jzmq
[storm@iZ25fnur5jkZ dependence]$./configure
[storm@iZ25fnur5jkZ dependence]$make
[storm@iZ25fnur5jkZ dependence]$make install
```

安装 Python 2.6.6(使用 root 账号，目录不相干)：

```
[storm@iZ25fnur5jkZ dependence]yum install python-devel
```

2. 安装 Storm

创建 storm 目录，用以安装 Storm：

```
[storm@iZ25fnur5jkZ shaka]$ cd /home/storm
[storm@iZ25fnur5jkZ shaka]$ mkdir storm
[storm@iZ25fnur5jkZ shaka]$ cd storm
```

从官网下载 Storm 安装包，下载地址为：

```
wget http://www.apache.org/dyn/closer.lua/storm/apache-storm-0.9.2-incubating/
apache-storm-0.9.2-incubating.tar.gz
```

解压 Storm 安装包：

```
[storm@iZ25fnur5jkZshaka]$ tar -xvf
apache-storm-0.9.2-incubating.deploy.tar.gz
```

配置环境变量：

```
vim ~/.bash_profile
export STORM_HOME=/home/shaka/storm/apache-storm-0.9.2-incubating
export PATH=$PATH:$STORM_HOME/bin
```

重载环境变量：

```
[storm@iZ25fnur5jkZ shaka]$  . ~/.bash_profile
```

修改配置文件 storm.yaml：

```
########## These MUST be filled in for a storm configuration
storm.zookeeper.servers:
     - "zookeeper-hostname01"
    # - "zookeeper-hostname02"
    # - "zookeeper-hostname03"
nimbus.host: "storm-node-01"
storm.local.dir:
"/home/storm/storm/apache-storm-0.9.2-incubating/topology"
ui.port: 8080
```

启动 Storm(在此之前，要启动 Zookeeper)：

```
[storm@iZ25fnur5jkZ shaka]$storm nimbus &
[storm@iZ25fnur5jkZ shaka]$storm supervisor &
[storm@iZ25fnur5jkZ shaka]$storm ui &
```

10.3.8 查看 Storm UI

启动 Storm 后，可以通过 http://{nimbus host}:8080 进行查看，如图 10-12 所示。

Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.2-incubating	5m 37s	1	0	4	4	0	0

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks

Supervisor summary

Id	Host	Uptime	Slots	Used slots
cd4d451a-0804-455f-a04e-a5254a3e16ac	hadoop	4m 2s	4	0

Nimbus Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx768m
drpc.invocations.port	3773
drpc.port	3772

图 10-12 Storm UI

10.3.9 搭建 Storm 集群

扩展 Storm 新节点。

将 shaka/storm 下的软件包完全一样地复制到新的机器上。

配置文件不需要修改，因为要共用 ZK 和 Nimbus。

配置环境变量：

```
vim ~/.bash_profile
export STORM_HOME=/home/shaka/storm/apache-storm-0.9.2-incubating
export PATH=$PATH:$STORM_HOME/bin
```

启动 supervisro &的方法：

```
[storm@iZ25fnur5jkZ shaka]$storm supervisor &
```

至此，集群版的 Storm 的安装完成。
一个集群模式的 Storm 就搭建起来了。

10.3.10　Storm 系统的操作实践

(1)　查看 Storm 命令的方法：

```
[storm@hadoop-nn apache-storm-0.9.2-incubating]$ storm help

Commands:
        activate
        classpath
        deactivate
        dev-zookeeper
        drpc
        help
        jar
        kill
        list
        localconfvalue
        logviewer
        nimbus
        rebalance
        remoteconfvalue
        repl
        shell
        supervisor
        ui
        version

Help:
        help
        help <command>
```

(2)　提交 Storm Topology：

```
storm jar mycode.jar storm.MyTopology arg1 arg2 ..
```

mycode.jar 是包含 Topology 实现代码的 JAR 包，storm.MyTopology 的 main 方法是 Topology 的入口，arg1、arg2 等为 main 方法参数。

(3)　列出 StormTopology：

```
[storm@iZ25fnur5jkZ shaka]$ storm list
```

(4)　停止 Storm Topology：

```
[storm@iZ25fnur5jkZ shaka]$ storm kill {topologyname}
```

(5)　查看版本信息：

```
[storm@hadoop-nn apache-storm-0.9.2-incubating]$ storm version
0.9.2-incubating
```

10.3.11 Storm WordCount(写 RDB)

Ant 编译的 Start 工程，主要实现 WordCount 功能，同时将最新结果更新到 MySQL 数据库中。Storm 做词频统计 Topology 的主要过程如图 10-13 所示。

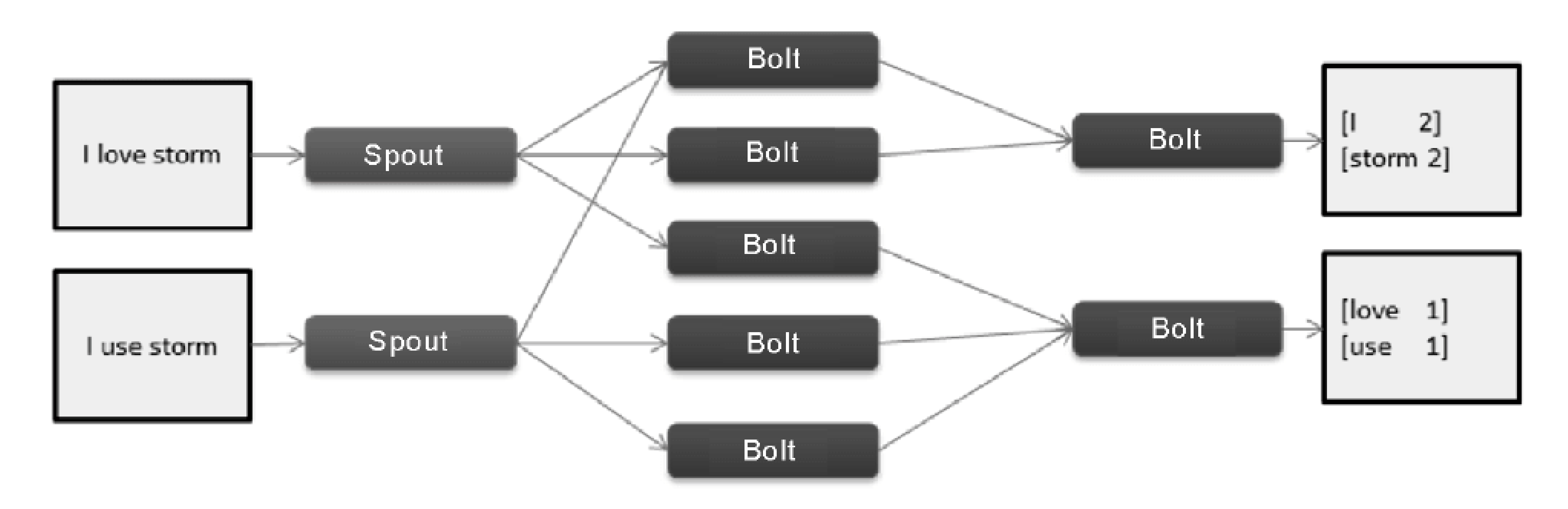


图 10-13 Storm 做词频统计 Topology 的主要过程

Storm 读 Kafka 需要的 JAR 包：

```
storm-kafka-0.9.2-incubating.jar
kafka_2.9.2-0.8.0-beta1.jar
metrics-core-2.2.0.jar
scala-library-2.9.2.jar
```

这几个 JAR 包在 Kafka 项目中能找到。放在 Storm 的 lib 下。

1. 主要实现类

WordCountTopology.java：

```
public static void main(String[] args) throws Exception {

    //Topology 创建
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("spout", new RandomSentenceSpout(), 1);
    builder.setBolt(
      "split", new SplitSentence(), 1).shuffleGrouping("spout");
    builder.setBolt("count", new WordCount(), 1)
      .fieldsGrouping("split", new Fields("word"));
    Config conf = new Config();
    conf.setDebug(true);
    //本地模式启动
    if (args!=null && args.length>0) {
        conf.setNumWorkers(1);
        StormSubmitter.submitTopology(
          args[0], conf, builder.createTopology());
    } else { //分布式模式
        conf.setMaxTaskParallelism(3);
        StormSubmitter.submitTopology(
          "word-count", conf, builder.createTopology());
    }
}
//其中 builder.setSpout("spout", new RandomSentenceSpout(), 1)设定数据喷发
//方式，本例子采用随机产生单词的方式。
//builder.setBolt("split", new SplitSentence(), 1)
//.shuffleGrouping("spout")指定分词方法。
```

```
//builder.setBolt("count", new WordCount(), 1).fieldsGrouping(
//"split", new Fields("word"))指定单词统计后的存储目的地。
```

RandomSentenceSpout 类的具体实现：

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;
    @Override
    public void open(Map conf, TopologyContext context,
      SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }
    @Override
    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[] { "the cow jumped over the moon",
          "an apple a day keeps the doctor away",
          "four score and seven years ago",
          "snow white and the seven dwarfs",
          "i am at two with nature" };
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }
    @Override
    public void ack(Object id) {}
    @Override
    public void fail(Object id) {}
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

RandomSentenceSpout 类的主要作用，是随机产生句子。其中最重要的为 nextTuple 函数，可以看到，每次都会随机地从字符串数组中选中一条记录进行发送。作为一个字段进行发送，字段名为 word。

SplitSentence 类的作用是对句子进行分词操作，本操作通过 splitsentence.py 脚本完成。

SplitSentence 类的具体代码如下：

```
public static class SplitSentence extends ShellBolt
  implements IRichBolt {
    public SplitSentence() {
        super("python", "splitsentence.py");
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
    @Override
    public Map<String, Object> getComponentConfiguration() {
        return null;
    }
}
```

Splitsentence.py 的内容如下：

```
import storm
```

```
class SplitSentenceBolt(storm.BasicBolt):
    def process(self, tup):
        words = tup.values[0].split(" ")
        for word in words:
            storm.emit([word])
SplitSentenceBolt().run()
```

很明显，该 Python 脚本以空格作为分隔符号对输入的字符串进行切分，每个单词作为下一个阶段的处理单元。

WordCount 类的具体实现代码如下：

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null)
            count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
        JMysql gmysql = new JMysql();
        JTimer gtime = new JTimer();
        String host = "112.126.71.xxx";
        String db = "shaka";
        String user = "shaka";
        String passwd = "teststorm";
        int ret = gmysql.initConnect(host, db, user, passwd);
        if (0 != ret)
        {
            System.out.println("create db conncet error.");
            return;
        }
        String sql = "select count(*) from t_stage_wordcount where word='"
                     + word + "'";
        boolean re = gmysql.exeSQL_Exist(sql);
        String updatetime = gtime.getCurrentTime();
        if (true == re) {
            //累加处理
            sql = "update  t_stage_wordcount set updatetime='" + updatetime
              + "', count="+count +" where word='" + word + "'";
        } else {
            sql =
            "insert into  t_stage_wordcount(word,count,updatetime) values('"
            + word + "'," + count + ",'" + updatetime + "');";
        }
        ret = gmysql.exeSQL_DUI(sql);
        ret = gmysql.freeConnect();
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        //从上一阶段过滤的字段
        declarer.declare(new Fields("word", "count"));
    }
}
```

单词统计的主要逻辑是：对于新词，直接执行插入操作，如果是已存在的单词，则直接更新单词次数。

JMysql 为封装的 MySQL 操作类，代码如下：

```
public class JMysql {
    //member variables
    private Connection conn = null;
    public int initConnect(String host, String db, String user, String passwd)
    {
        try {
            String driver    = "com.mysql.jdbc.Driver";
            String connector = "jdbc:mysql://" + host + ":3306/" + db
            + "?characterEncoding=utf-8&allowMultiQueries=
              true&autoReconnect=true&failOverReadOnly=false";
            //create db connect.
            Class.forName(driver);
            conn = DriverManager.getConnection(connector, user, passwd);
            return 0;
        } catch (Exception e) {
            e.printStackTrace();
            return 1;
        }
    }
    public int  freeConnect()
    {
        try {
            conn.close();
            return 0;
        } catch (SQLException e) {
            e.printStackTrace();
            return 1;
        }
    }
    public int exeSQL_DUI(String sql)
    {
        Statement stmt = null;
        int ret = -1;
        try {
            stmt = (Statement)conn.createStatement();
            ret = stmt.executeUpdate(sql);
            stmt.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return ret;
    }
    public ArrayList exeSQL_S(String sql)
    {
        //set variables
        ArrayList list    = new ArrayList();
        Statement stmt    = null;
        ResultSet ret     = null;
        //get result
        try {
            stmt = (Statement)conn.createStatement();
            ret = stmt.executeQuery(sql);
            while (ret.next())
            {
                list.add(ret.getString(1) + "," + ret.getString(2));
            }
            stmt.close();
            return list;
        } catch (SQLException e) {
```

```
            e.printStackTrace();
            return null;
        }
    }
    public boolean exeSQL_Exist(String sql)
    {
        //set variables
        ArrayList list     = new ArrayList();
        Statement stmt     = null;
        ResultSet rst      = null;
        boolean  ret       = false;
        //get result
        try {
            System.out.println(sql);
            stmt = (Statement)conn.createStatement();
            rst = stmt.executeQuery(sql);
            int count = -1;
            while(rst.next()) {
                count = rst.getInt(1);
            }
            if(0 == count) { ret = false; } else { ret = true; }
            stmt.close();
            return ret;
        } catch (SQLException e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

启动 topology 程序：

```
[storm@hadoop-nn starter]$ storm jar dist/topology-0.0.1.jar
starter.WordCountTopology
```

2. 结果

数据截图如图 10-14 所示。

```
1 select * from t_stage_wordcount order by count desc;
```

t_stage_wordcount (3×27)

word	count	updatetime
the	2813	2016-04-09 23:26:28
seven	1344	2016-04-09 23:26:28
and	1344	2016-04-09 23:26:28
over	714	2016-04-09 23:26:20
moon	714	2016-04-09 23:26:21
cow	714	2016-04-09 23:26:20
jumped	714	2016-04-09 23:26:20
away	712	2016-04-09 23:26:27
doctor	712	2016-04-09 23:26:27
keeps	712	2016-04-09 23:26:26
a	712	2016-04-09 23:26:26
apple	712	2016-04-09 23:26:26
an	712	2016-04-09 23:26:26
day	712	2016-04-09 23:26:26
white	673	2016-04-09 23:26:28
snow	673	2016-04-09 23:26:28
dwarfs	673	2016-04-09 23:26:28
four	671	2016-04-09 23:26:28

图 10-14　结果图

10.3.12　Storm WordCount(从 Kafka 读取数据)

进一步改进，让数据的产生不是随机的，而是从 Kafka 的指定 topic 队列中获取的。其流程(构建图)如图 10-15 所示。

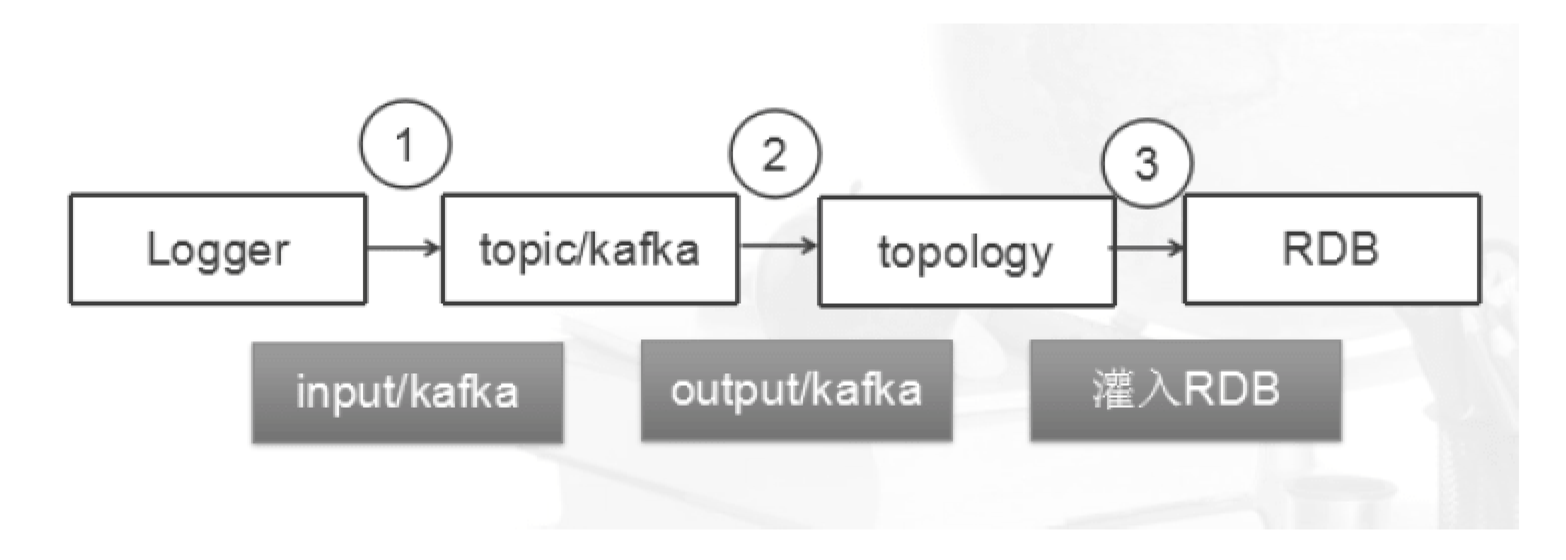


图 10-15　构建图

Topology 的构建过程：

```
public static void main(String[] args) throws Exception {

    TopologyBuilder builder = new TopologyBuilder();

    String Zklist     = "localhost:2181";
    String data_topic   = "mykafka";
    String data_zkpaht  = "/test/wordcount/model";
    String data_zkid    = "data_id";
    ZkHosts zkHosts = new ZkHosts(Zklist);
    SpoutConfig dataConfig   =
      new SpoutConfig(zkHosts, data_topic, data_zkpaht, data_zkid);
    dataConfig.scheme     = new SchemeAsMultiScheme(new StringScheme());
    dataConfig.fetchSizeBytes = 10000000;
    dataConfig.bufferSizeBytes= 10000000;

    //builder.setSpout("spout", new RandomSentenceSpout(), 1);
    builder.setSpout("spout", new KafkaSpout(dataConfig), 1);
    builder.setBolt("split", new SplitSentence(), 1)
                     .shuffleGrouping("spout");
    builder.setBolt("count", new WordCount(), 1)
                     .fieldsGrouping("split", new Fields("word"));

    Config conf = new Config();
    conf.setDebug(true);

    if (args != null && args.length > 0) {
        conf.setNumWorkers(1);
        StormSubmitter.submitTopology(
          args[0], conf, builder.createTopology());
    }
    else {
        conf.setMaxTaskParallelism(3);
        StormSubmitter.submitTopology(
          "s-word-count", conf, builder.createTopology());
    }
}
```

启动 topology 程序：

```
[storm@hadoop-nn starter]$ storm jar dist/topology-0.0.1.jar
starter.WordCountTopology

storm jar dist/topology-0.0.1.jar
starter.WordCountTopologyLoggerKafkaStormMysql
```

Storm UI 截图如图 10-16 所示。

Storm UI

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
s-word-count	s-word-count-7-1460217290	ACTIVE	1m 4s	4	1	4

Topology actions

Activate | Deactivate | Rebalance | Kill

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	1180	1040	0.000	320	0
3h 0m 0s	1180	1040	0.000	320	0
1d 0h 0m 0s	1180	1040	0.000	320	0
All time	1180	1040	0.000	320	0

Spouts (All time)

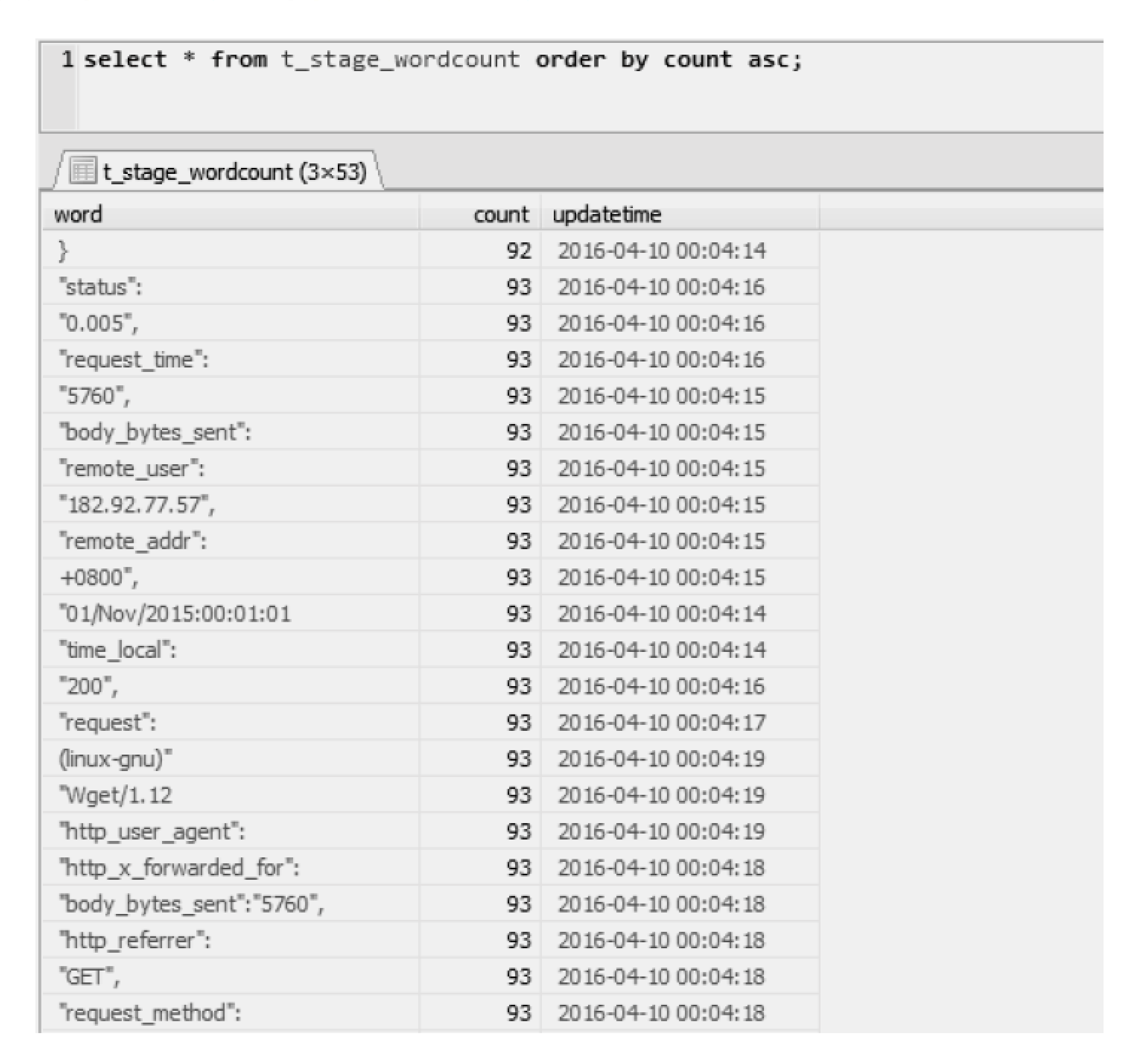
Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Last error
spout	1	1	60	60	0.000	0	0	

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Last error
__acker	1	1	0	0	0.000	0.111	180	0.222	180	0	
count	1	1	220	80	0.815	388.833	120	434.000	120	0	
split	1	1	900	900	0.000	0.000	40	38.000	20	0	

图 10-16　Storm UI 截图

MySQL 数据库的运行结果如图 10-17 所示。



```
1 select * from t_stage_wordcount order by count asc;
```

t_stage_wordcount (3×53)

word	count	updatetime
}	92	2016-04-10 00:04:14
"status":	93	2016-04-10 00:04:16
"0.005",	93	2016-04-10 00:04:16
"request_time":	93	2016-04-10 00:04:16
"5760",	93	2016-04-10 00:04:15
"body_bytes_sent":	93	2016-04-10 00:04:15
"remote_user":	93	2016-04-10 00:04:15
"182.92.77.57",	93	2016-04-10 00:04:15
"remote_addr":	93	2016-04-10 00:04:15
+0800",	93	2016-04-10 00:04:15
"01/Nov/2015:00:01:01	93	2016-04-10 00:04:14
"time_local":	93	2016-04-10 00:04:14
"200",	93	2016-04-10 00:04:16
"request":	93	2016-04-10 00:04:17
(linux-gnu)"	93	2016-04-10 00:04:19
"Wget/1.12	93	2016-04-10 00:04:19
"http_user_agent":	93	2016-04-10 00:04:19
"http_x_forwarded_for":	93	2016-04-10 00:04:18
"body_bytes_sent":"5760",	93	2016-04-10 00:04:18
"http_referrer":	93	2016-04-10 00:04:18
"GET",	93	2016-04-10 00:04:18
"request_method":	93	2016-04-10 00:04:18

图 10-17　MySQL 的运行结果

10.4　小　　结

实时数据系统在大数据领域得到了迅速的发展，Storm 在活跃社区成员的不断贡献下，性能、速度、稳定性等都得到了极大的提升。同时，实时计算引擎除了本章介绍的 Storm，还涌现出多种其他优秀的实时计算引擎，如 Spark Streaming、S4 等。这几种实时计算技术都有自己的特点及适用场景。

本章通过基本理论的介绍以及实例的操作，让读者对实时数据系统有更直接的认识，了解业界在实时计算领域的常用技术架构及编程实现的过程。

参 考 文 献

[1] http://hadoop.apache.org/docs/r2.6.0/

[2] 怀特. Hadoop 权威指南[M]. 2 版. 曾大聃，周傲英，译. 北京：清华大学出版社，2010.

[3] Hadoop 新 MapReduce 框架 YARN 详解. IBM 开发社区. 2013.

[4] http://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html

[5] Hadoop 官网：http://hadoop.apache.org/docs/r2.7.2/

[6] Kafka 官网：http://kafka.apache.org/

[7] Storm 官网：http://storm.apache.org/